Writing Converters for the Function Convert Package

Barton Willis

February 25, 2026

Contents

1 Introduction and prerequisites

Maxima's `function convert` package [1] performs semantic function-to-function conversions on Maxima [2] expressions. This tutorial explains how to extend the package's mathematical capabilities by writing your own converters. To follow this tutorial, you should be comfortable with:

- using Maxima and the `function convert` package, and
- programming Maxima in Common Lisp.

This guide is written for people who want to create new converters. It focuses on the writing converters: patterns, idioms, and design principles. Here we include only the API elements that matter when you are authoring a converter.

To install this package, place the file '`file_convert.lisp`' in a directory on Maxima's search path and load it with '`load("function_convert");`'. Eventually, this package might be migrated to Maxima's /src folder–when this happens, loading the package will not be needed.

We begin with the basic usage of the `function convert` package and an overview of its main features.

Source Note Portions of this tutorial are taken from, or adapted from, the project's `README.md` on GitHub [1], with the permission of the author.

2 Usage and Features

The most basic use of the package is a converter that replaces each subexpression of a Maxima expression of the form $\operatorname{erfc}(X)$ with by $1 - \operatorname{erf}(X)$, where and are the error functions and X may be any expression. This converter is built into the `function convert` package. To use it, enter:

```
(%i1) function_convert(erfc = erf, erfc(x));
(%o1)                              1 - erf(x)
```

Of course, the input can be any expression, including nested calls to erf; for example

```
(%i2) function_convert(erfc = erf, x*erfc(x + erfc(z)));
(%o2)                        x (erf(erf(z) - x - 1) + 1)
```

The first argument '`erfc = erf`' is called a *conversion rule* (or simply a 'rule'), the left-hand side is the *source function* and the right-hand side '`erf`' is called the *target function*. The rule '`erfc = erf`' does not literally replace the source function with the target function, but it instead applies the identity $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$ to an expression, replacing the left-hand side with the right-hand side.

The name of the target function is best thought of a mnemonic for recalling the identity $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$, not as a literal function. Actually, the name of the target function could be any Maxima symbol; for example, the rule could be named '`erfc = erfc_to_erf`'.

2.1 Algorithm and Implementation

The function '`function_convert`' walks an expression tree and replaces function calls according to well-defined rules. In that sense, it is straightforward code of a kind repeated many times in Maxima. It is not a pattern matcher or a general rewrite engine.

The package is implemented in Common Lisp. It has been tested using Clozure Common Lisp [3] version 1.13 and SBCL [4] version 2.4.7.

2.2 Naming and the Alias System

We've seen that the name of the target function is best thought of an alias for the identity being applied. The `function convert` package has an alias system allows the source function to be a mnemonic as well. This feature is useful for naming a converter for an identity such as

$$\Gamma(z)\,\Gamma(1-z) = \pi/\sin(\pi z)\,. \tag{1}$$

Here the left-hand side of this identity is a product, so the source function for this identity must be multiplication (internally '`mtimes`'), not the gamma function. But the rule '`"*" = sin`' is hard to remember and unnatural. The alias system allows the converter for this identity to have a much more natural name. For the identity in Eq. (1), using the alias system we can name this rule '`gamma = sin`', for example. Actually, this converter built-in to the `function convert` package:

```
(%i1) function_convert(gamma = sin, gamma(x)*gamma(1-x));
(%o1) %pi/sin(%pi*x)

(%i2) function_convert(gamma = sin, gamma(2 + x)*gamma(1-x));
(%o2) gamma(1-x)*gamma(x+2)
```

A macro takes care of the messy details of defining an alias, along with other housekeeping duties.

2.3 Built-in converters

A converter may be marked as built-in. Built-in converters cannot be removed using any user-level function. For example, the converter '`erfc = erf`' is built-in; attempting to remove it with '`delete_converter`' produces an error:

```
(%i1) delete_converter(erfc = erf);
Cannot delete built-in converter (%ERFC → %ERF).
```

2.4 Self-documentation feature

Every converter function should have a docstring. The docstring is available to the user via the function '`describe_converter`'; for example

```
    (%i1) describe_converter(sinc = sin);
Converter: %SINC = %SIN
Function: #<FUNCTION FUNCTION-CONVERTER-%SINC-%SIN>
Type: built-in

Docstring:
  Convert sinc(x) into sin(x)/x.
```

If a user-package properly includes a docstring for a converter, that documentation will be available via the function ‘describe_converter’ once the package is loaded. The name of the converter, in this case ‘FUNCTION-CONVERTER-%SINC-%SIN’, is automatically generated by a macro that is used to define a converter–the function name of a converter is unlikely to be useful to a user.

2.5 Class Keys and Class-Based Dispatch

Converter functions often need to apply the same transformation to an entire family of related operators. For example, a converter that rewrites trigonometric functions into exponential functions should apply uniformly to all six trigonometric functions. Writing six separate converters is tedious and error-prone. The *class key* mechanism provides a solution.

Purpose of Class Keys A *class key* is a symbolic tag (such as ‘:trig’ or ‘:hyperbolic’) that represents a category of operators. The converter system uses these tags to perform *class-based dispatch*: a single converter registered for a class key is automatically applied to every operator belonging to that class.

Class keys classify operators, not expression types. For example, the class ‘:algebraic’ contains the operators ‘mplus’, ‘mtimes’, and ‘mexpt’. If the head operator of an expression belongs to the class ‘:algebraic’, that does not imply that the expression itself is algebraic.

The Class Table The mapping from class keys to operators resides in the global ‘*converter-class-table*’. This table is

```
(defparameter *converter-class-table*
  '((:trig        . (%sin %cos %tan %sec %csc %cot))
    (:hyperbolic  . (%sinh %cosh %tanh %sech %csch %coth))
    (:inv_trig    . (%asin %acos %atan %asec %acsc %acot))
    (:inv_hyperbolic . (%asinh %acosh %atanh %asech %acsch %acoth))
    (:exp            . (mexpt))
    (:gamma_like     . (%gamma %beta %binomial %double_factorial
                        mfactorial $pochhammer))
    (:bessel  . (%bessel_j %bessel_y %bessel_i %bessel_k
                 %hankel_1 %hankel_2 %airy_ai %airy_bi
                 %airy_dai %airy_dbi))
    (:algebraic . (mplus mtimes mexpt))
    (:inequation . (mequal mlessp mleqp mnotequal mgreaterp mgeqp
                    $notequal $equal))
    (:logarithmic . (%log)))
  "Mapping from class keys to lists of operator symbols.")
```

Each entry associates a class key with the list of operators that belong to that class. The system assumes that *an operator belongs to exactly one class*. This ensures unambiguous dispatch: when the converter system encounters an operator, it can determine its class uniquely and apply the appropriate class-level converter.

How Class Dispatch Works When the converter system processes an expression, it identifies the head operator of the form. It then:

(a) looks up the operator in '`*converter-class-table*`';
(b) determines the corresponding class key (e.g., '`:trig`');
(c) checks whether a converter is registered for that class;
(d) if so, applies that converter to the entire expression.

Why Class Keys Matter Class keys provide:

- **Abstraction**: one converter per conceptual family.
- **Extensibility**: adding a new operator to a class requires no changes to existing converters.
- **Uniformity**: guarantees consistent behavior across related operators.
- **Clarity**: the class table documents the structure of the operator space.

2.6 Chaining multiple conversions

To apply two or more conversions, put the converters into a list; for example

```
(%i1) function_convert(['sinc = 'sin, 'sin = 'exp], sinc(x)^2);
(%o1) -((%e^(%i*x)-%e^-(%i*x))^2/(4*x^2))
```

Converters are applied left-to-right.

Chaining multiple conversions is handled by the function '`function_convert`'; a writer of a converter doesn't need to do anything to allow chaining to work.

2.7 Implicit Dispatch via Breadth-First Search

The converter system automatically discovers multistep conversion chains. For example, if a converter is defined from f to g and another from g to h, the system is able to convert f to h without any additional user-defined rule. In other words, there is no need to define a direct converter f to h.

This behavior is enabled by a breadth-first search (BFS) over the converter graph. [5] The search locates the shortest available path between two operators and composes the corresponding converters. Writers of converters only need to know that `function convert` uses BFS, not anything about the internals of this mechanism.

2.8 Errors and Debugging

When a converter is redefined, Maxima will issue a warning, and when a converter doesn't exist, Maxima returns the input unchanged, but doesn't issue a warning; for example

```
(%i2) function_convert(f = g, a+b);
(%o2)                              b + a
```

When writing a new converter, if a converter doesn't trigger, the problem is likely due to either a typo or some confusion between Maxima nouns and verbs. If a converter doesn't trigger correctly, try tracing the Common Lisp function '`lookup-converter`'. From there, you might be able to diagnose the problem.

3 Writing converter functions

A macro '`define-function-converter`' takes care of many of the housekeeping details in writing a converter, including naming the internal function that applies the rule, optionally marking the converter as built-in, and gathering the docstring for the converter to use a user-documentation. The macro also automatically handles the details needed to make the alias system work.

The function '`function_convert`' does the work of mapping the converter over the expression tree; this allows the converter code to be relatively simple. We'll begin by examining the source code of the '`erfc = erf`' converter.

3.1 The erfc to erf converter

The source code for the erfc to erf converter is

```
(define-function-converter (%erfc %erf) (op x)
    :builtin
  "Convert erfc(x) into 1 - erf(x)."
  (declare (ignore op))
  (let ((z (car x)))
    (sub 1 (ftake '%erf z))))
```

The arguments '`%erf %erfc`' are the source and target functions, respectively. For this converter, the argument '`op`' is bound to the source function '`%erfc`', but this converter doesn't use this argument, so the converter code should declare to ignore it. The argument '`x`' is a Common Lisp list of the argument to the function '`%erfc`'.

The optional keyword argument '`:builtin`' declares that this converter cannot be deleted using user-level functions. To mark a rule as built-in, the keyword '`:builtin`' must be placed before the docstring. For a non-built-in rule, omit the '`:builtin`' keyword. The docstring is optional, but important because supplies the converter with documentation.

The mathematical part of the code, that is

```
  (let ((z (car x)))
    (sub 1 (ftake '%erf z)))
```

implements the identity $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$. Notice that since '`x`' is a Common Lisp list of the argument to '`%erfc`', it is necessary to apply '`%erf`' to the car of '`x`', and not to '`x`' itself. Alternatively, if you prefer, this bit of code can be changed to

```
      (sub 1 (fapply '%erf x))
```

3.2 Alias Dispatch

As we discussed before, both the source and the target functions of a converter can be given an alias. This can be useful when the actual source function is a poor mnemonic for the converter name.

Again, the macro ‘define-function-converter’ takes care of the housekeeping for defining an alias. The best way to explain how define a converter that uses alias dispatch is by an example. Here we give a converter that expands explicit powers, but not products of sums. The source function must be ‘mexpt’, but we’ll make ‘power = expand’ an alias for "^" = expand. This is how it is done:

```
(define-function-converter ((mexpt $expand) ($power $expand)) (op x)
  ($expand (fapply 'mexpt x)))
```

A rule can have at most one alias.

There are two ways to invoke this converter:

```
(%i1) function_convert("^" = expand, (a+b)*x + (1+x)^2);
(%o1) x^2+(b+a)*x+2*x+1

(%i2) function_convert(power = expand_powers, (a+b)*x + (1+x)^2);
(%o2) x^2+(b+a)*x+2*x+1
```

3.3 Class Key Converter Dispatch

Class-key converters allow a single definition to handle an entire family of related operators, for example all trigonometric functions. The following example illustrates this idea using the trigonometric class ‘:trig’. Any operator listed under the ‘:trig’ class key in ‘*converter-class-table*’ will be handled by the same converter, without the need to define six separate rules.

Specifically, we’ll show how to define a rule that rewrites all trigonometric functions in terms of sine and cosine. For this converter, we finally have a reason for the ‘op’ argument — this argument will be bound to one of the six trigonometric functions. The main code uses a ‘case’ statement to handle each of the six cases. The code is

```
(define-function-converter (:trig $sin_cos) (op x)
  :builtin
  ;; Convert all six trigonometric functions to sin/cos form.
  (let ((z (first x)))
    (case op
      (%sin (ftake '%sin z))
      (%cos (ftake '%cos z))
      (%tan (div (ftake '%sin z) (ftake '%cos z)))
      (%sec (div 1 (ftake '%cos z)))
      (%csc (div 1 (ftake '%sin z)))
      (%cot (div (ftake '%cos z) (ftake '%sin z)))
      (t (ftake op z)))))
```

This converter is registered for the class key ‘:trig’, not for any individual operator. When the dispatch system encounters an expression such as ‘((%tan) x)’ or ‘((%csc) x)’, it determines that the operator belongs to the ‘:trig’ class and invokes the converter above.

The body of the converter receives two arguments: the operator `'op'` and its argument list `'x'`. The converter then rewrites each trigonometric function into its sine-cosine form. Because the class key handles the grouping, the converter itself contains only the mathematical logic; it does not need to know how many operators belong to the class or where they appear in the system.

3.4 A more advanced converter

In this section, we show to to build a converter that rewrites explicit products of the form $x\,\mathrm{signum}(x)$ as $|x|$. The converter does not rewrite $\mathrm{signum}(x)$ itself, because the pattern $x\,\mathrm{signum}(x)$ is not a subexpression of $\mathrm{signum}(x)$; only explicit products matching that pattern are transformed — this is what makes writing the converter somewhat tricky. The converter logic needs to scan the arguments of each product and gather the terms that have signum as their head operator. For each of these terms, the converter does a rational substitution (`'ratsimp'`) of the form `'ratsubst(abs(s), s*signum(s), e)'` on the expression `'e'`, that must be a product. The definition is:

```
(define-function-converter ((mtimes mabs) (%signum mabs)) (op x)
  :builtin
  "Convert subexpressions of the form X*signum(X) into abs(X).  This converter
does not rewrite signum(X) itself, since X*signum(X) is not a subexpression
of signum(X); only explicit products matching that pattern are transformed."
  (let* ((e  (fapply op x))
         (ll (xgather-args-of e '%signum)))
    (dolist (lx ll)
      (let ((s (car lx)))
        (setq e ($ratsubst (ftake 'mabs s)
                           (mul s (ftake '%signum s))
                           e))))
    ($expand e 0 0)))
```

How the converter works

(a) The converter receives the head operator `'op'` and its argument list `'x'`. The call `'(fapply op x)'` reconstructs the full expression `'e'`.
(b) The function `'xgather-args-of'` scans `'e'`. for occurrences of `'%signum'` and returns a list of matches. Each element of this list contains the argument X of a subexpression $\mathrm{signum}(X)$. The `function convert` package defines the function `'xgather-args-of'`.
(c) For each such argument s, the converter constructs the product $s\,\mathrm{signum}(s)$ and replaces it with $|s|$ using `'ratsubst'`. Only explicit products are rewritten; no algebraic inference is performed.
(d) After all substitutions, the expression is resimplified with a call to `'expand'`.

This converter is intentionally conservative: it rewrites only the literal pattern $X\,\mathrm{signum}(X)$, never the function signum itself.

4 Regression Tests for function convert

This section describes how to write regression tests for the 'function_convert' package and how to run them.

Test File Format A regression test file is an ordinary Maxima batch file. Each test consists of *exactly two lines*:

```
input;
expected output$
```

The first line is the Maxima expression to evaluate, and the second line is the expected result. Terminating the input line with a semicolon and the expected output with a dollar sign clarifies the intent of each line, but the line terminators can be either a semicolon or a dollar sign, if you prefer. A file can have as many tests as needed. Here is an example of one test:

```
function_convert('cos = 'exp, cos(x));
(%e^(%i*x)+%e^-(%i*x))/2$
```

Running the Tests If your tests are in a file 'rtest_my_function_convert', to run the regression tests, use:

```
batch(rtest_my_function_convert, 'test);
```

The command 'batch' with an optional second argument of 'test', loads the test file, evaluates each input expression, compares the actual output with the expected output, and reports any mismatches. A successful run prints no errors.

What to Test

- bogus, unexpected (boolean values, strings, or . . .), or invalid input,
- complex numbers,
- nested expressions and multiple occurrences the source function,
- converters applied to expressions containing unknown functions.

Writing Good Tests A good regression test is minimal, documents the intended behavior of the converter system, and isolates a single behavior. If a converter is based on an identity, each such converter should have an explicit test of the identity; here is an example that tests the identity $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$:

```
erf(x) + function_convert(erfc = erf, erfc(x));
1$
```

5 Best Practices for Writing Converters

The guidelines below help ensure that new converters integrate smoothly into the `function convert` system.

- **Base converters on identities.** A converter should be based on a mathematical identity. It should not change the meaning of an expression outside its intended domain. If a transformation is only valid under certain assumptions, the converter should check them explicitly or decline to apply the transformation. It is discouraged, but possible, to write a converter that, for example, replaces every product by the number 42.
- **Include a docstring.** Each converter should have a docstring so that `function convert` is self-documenting. If a converter requires a domain restriction or mathematical convention, document it. Clear documentation prevents misuse and helps future contributors maintain the system.
- **Use clear, descriptive names.** Converter names or aliases should reflect the mathematical transformation they perform. This helps users understand the converter registry and simplifies debugging.
- **Keep converters small and focused.** A converter should perform one clear transformation, such as rewriting $\operatorname{erfc}(x)$ as $1 - \operatorname{erf}(x)$ or converting $\tan(x)$ to $\sin(x)/\cos(x)$. Small, focused converters compose more reliably and are easier to test and debug.
- **Dispatch on functions, not patterns.** Converters operate on function calls, not syntactic patterns. Let the converter receive the arguments directly and compute the replacement semantically. This avoids brittle rules and reduces unintended matches.
- **Ensure termination.** Converters should not create expressions that trigger themselves again unless the transformation is guaranteed to converge.
- **Write regression tests.** Each converter should have regression tests covering typical inputs, edge cases, and expressions where the converter should *not* apply. Tests help maintain stability as the system evolves.

6 Miscellaneous

We end with a few comments about the `function convert` package that are not directly related to the process of writing converter functions.

6.1 Motivation

Many systems (Maple [6], Mathematica [7], SymPy [8]) provide built-in expansions or rewrite mechanisms, but Maxima uses an alphabet soup of functions that perform semantic function-to-function conversions; examples include `‘makefact’` and `‘makegamma’`. In other cases, transformations are controlled by option variables — for example, `‘expintrep’`.

These names are easy to forget and are not always easy to locate in the user documentation. The `function convert` package may offer a simple, uniform, and user-extensible way to perform such conversions.

6.2 Using `‘defrule’` as an alternative to `function convert`

Maxima’s pattern-based `‘defrule’` tool is an alternative to using `function convert`. A simple example:

```
(%i1) matchdeclare(aa,true)$

(%i2) defrule(sinc_rule, sinc(aa), sin(aa)/aa);
(%o2) sinc_rule:marrow(sinc(aa),sin(aa)/aa)

(%i3) apply1(sinc(sinc(x)), sinc_rule);
(%o3) (x*sin(sin(x)/x))/sin(x)
```

This method works well, especially for single-use function-to-function conversions. However, a '`kill(all)`' removes all rules defined by '`defrule`', and the approach still depends on an alphabet soup of auxiliary functions. Changing Maxima to allow rules that cannot be removed by '`kill(all)`' is, of course, a small matter of programming [9], but it would violate a long-standing design principle: users must always be able to return the system to a clean, rule-free state.

Additionally, it is difficult to define rules that need to match two or more terms in a sum or product. The `function convert` package has several converters that do just that. Often these converters internally rely on `ratsubst` to do semantic substitutions — something that is difficult for a rule defined by '`defrule`' to do. The package has at least one built-in rule that does just that — it difficult to duplicate using '`defrule`':

```
(%i1) function_convert('gamma = 'sin, 20252*x*gamma(x)*gamma(1-x) < %pi);
(%o1) (20252*%pi*x)/sin(%pi*x) < %pi
```

About the Author This manual was written by Barton Willis, a long-time contributor to the Maxima computer algebra system and the author of the `function convert` package. He has worked with symbolic computation, mathematical software, and technical documentation for many years.

Colophon This manual was typeset using the `pdflatex` engine with the `extarticle` document class. The body text is set in a serif typeface chosen for clarity in technical prose. Code fragments and Maxima sessions are rendered in a monospaced font with consistent verbatim handling. Bibliographic data were managed using `bibtex` using the `IEEEtran` style.

All examples were executed using Maxima from the current development branch together with the `function convert` package also from the current development branch.

Portions of this document were adapted from the project's `README.md` on GitHub, with the permission of the author.

This manual is intended as a long-term reference for users and developers who wish to extend the mathematical capabilities of the `function convert` package by writing custom converters.

References

[1] B. Willis, "function_convert: Semantic function-to-function conversions for maxima," https://github.com/barton-willis/function_convert, 2026, gitHub repository.

[2] Maxima Development Team, *Maxima, a Computer Algebra System*, Maxima, 2026, version 5.47.0 or later. [Online]. Available: https://maxima.sourceforge.io/

[3] Clozure Associates, *Clozure Common Lisp*, Clozure, 2026, version 1.13 or later. [Online]. Available: https://ccl.clozure.com/

[4] SBCL Development Team, *SBCL: Steel Bank Common Lisp*, Steel Bank Common Lisp, 2026, version 2.x. [Online]. Available: https://www.sbcl.org/

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. MIT Press, 2022.

[6] Maplesoft, “The convert function,” https://www.maplesoft.com/support/help/maple/view.aspx?path=convert, accessed 2026-02-23.

[7] W. Research, “Rules,” https://reference.wolfram.com/language/guide/Rules.html, accessed 2026-02-23.

[8] S. D. Team, “Term rewriting,” https://docs.sympy.org/latest/modules/rewriting.html, accessed 2026-02-23.

[9] B. A. Nardi, *A Small Matter of Programming: Perspectives on End User Computing*. Cambridge, MA: MIT Press, 1993.