LaTeX2 Functional Interfaces to LaTeX3 Programming Layer

Jianrui Lyu (tolvjr@163.com)
https://github.com/lvjr/functional

Version 2022A (2022-03-10)

LaTeX3 programming layer (expl3) is very powerful for advanced users, but it is a little complicated for normal users. This functional package aims to provide intuitive LaTeX2 functional interfaces for it.

Although there are functions in LaTeX3, the evaluation of them is from outside to inside. With this package, the evaluation of functions is from inside to outside, which is the same as other programming languages such as `JavaScript` or `Lua`. In this way, it is rather easy to debug code too.

Note that many paragraphs in this manual are copied from the documentation of expl3.

Contents

Chapter 1

Overview of Features

1.1 Evaluation from Inside to Outside

We will compare our first example with a similar `Lua` example:

```
-- lua code --
function MathSquare (arg)
  local lTmpaInt = arg * arg
  return lTmpaInt
end
print(MathSquare(5))
print(MathSquare(MathSquare(5)))
```

```
\ExplSyntaxOn
\PrgNewFunction \MathSquare { m } {
  \IntSet \lTmpaInt { \IntEval { #1 * #1 } }
  \Result { \Value \lTmpaInt }
}
\ExplSyntaxOff
\MathSquare{5}
\MathSquare{\MathSquare{5}}
```

Both examples calculate first the square of 5 and produce 25, then calculate the square of 25 and produce 625. In contrast to `expl3`, this `functional` package does evaluation of functions from inside to outside, which means composition of functions works like othe programming languages such as `Lua` or `JavsScript`.

You can define new functions with `\PrgNewFunction` command. To make composition of functions work as expected, every function *must not* insert directly any token to the input stream. Instead, a function *must* pass the result (if any) to `functional` package with `\Result` command. And `functional` package is responsible for inserting result tokens to the input stream at the appropriate time.

To remove space tokens inside function code in defining functions, you'd better put function definitions inside `\ExplSyntaxOn` and `\ExplSyntaxOff` block. Within this block, `~` is used to input a space.

At the end of this section, we will compare our factorial example with a similar `Lua` example:

```
-- lua code --
function Factorial (n)
  if n == 0 then
    return 1
  else
    return n * Factorial(n-1)
  end
end
print(Factorial(4))
```

```
\ExplSyntaxOn
\PrgNewFunction \Factorial { m } {
  \IntCompareTF {#1} = {0} {
    \Result {1}
  }{
    \Result { \IntMathMult {#1} { \Factorial { \IntMathSub{#1}{1} } } }
  }
}
\ExplSyntaxOff
\Factorial{4}
```

1.2 Group Scoping of Functions

In Lua language, a function or a condition expression makes a block, and the values of local variables will be reset after a block. For example

```
-- lua code --
local a = 1
print(a)          ---- 1
function SomeFun()
  local a = 2
  print(a)        ---- 2
  if 1 > 0 then
    local a = 3
    print(a)      ---- 3
  end
  print(a)        ---- 2
end
SomeFun()
print(a)          ---- 1
```

In `functional` package, a condition expression is in fact a function, and you can make every function become a group by setting `\Functional{scoping=true}`. For example

```
\Functional{scoping=true}
\ExplSyntaxOn
\IntSet \lTmpaInt {1}
\IntLog \lTmpaInt             % ---- 1
\PrgNewFunction \SomeFun { } {
  \IntSet \lTmpaInt {2}
  \IntLog \lTmpaInt           % ---- 2
  \IntCompareTF {1} > {0} {
    \IntSet \lTmpaInt {3}
    \IntLog \lTmpaInt         % ---- 3
  }{ }
  \IntLog \lTmpaInt           % ---- 2
}
\SomeFun
\IntLog \lTmpaInt             % ---- 1
\ExplSyntaxOff
```

Same as `expl3`, the names of local variables *must* start with `l`, while names of global variables *must* start with `g`. The difference is that `functional` package provides only one function for setting both local and global varianbles of the same type, by checking leading letters of their names. So for integer variables, you can write `\IntSet\lTmpaInt{1}` and `\IntSet\gTmpbInt{2}`.

The previous example will produce different result if we change variable from `\lTmpaInt` to `\gTmpaInt`.

```
\Functional{scoping=true}
\IntSet \gTmpaInt {1}
\IntLog \gTmpaInt              % ---- 1
\PrgNewFunction \SomeFun { } {
  \IntSet \gTmpaInt {2}
  \IntLog \gTmpaInt            % ---- 2
  \IntCompareTF {1} > {0} {
    \IntSet \gTmpaInt {3}
    \IntLog \gTmpaInt          % ---- 3
  }{ }
  \IntLog \gTmpaInt            % ---- 3
}
\SomeFun
\IntLog \gTmpaInt              % ---- 3
```

As you can see, the values of global variables will never be reset after a group.

1.3 Tracing Evaluation of Functions

Since every function in `functional` package will pass its return value to the package, it is quite easy to debug your code. You can turn on the tracing by setting `\Functional{tracing=true}`. For example, the tracing log of the first example in this chapter will be the following:

```
[I] \MathSquare{5}
        [I] \IntEval{5*5}
                [I] \Expand{\int_eval:n {5*5}}
                [0] 25
            [I] \Result{25}
            [0] 25
        [0] 25
    [I] \IntSet\lTmpaInt {25}
    [0]
        [I] \Value\lTmpaInt
        [0] 25
    [I] \Result{25}
    [0] 25
[0] 25
[I] \MathSquare{25}
        [I] \IntEval{25*25}
                [I] \Expand{\int_eval:n {25*25}}
                [0] 625
            [I] \Result{625}
            [0] 625
        [0] 625
    [I] \IntSet\lTmpaInt {625}
    [0]
        [I] \Value\lTmpaInt
        [0] 625
    [I] \Result{625}
    [0] 625
[0] 625
```

1.4 Definitions of Functions

Within `expl3`, there are eight commands for defining new functions, which is good for power users.

```
\cs_new:Npn
\cs_new_nopar:Npn
\cs_new_protected:Npn
\cs_new_protected_nopar:Npn
\cs_new:Nn
\cs_new_nopar:Nn
\cs_new_protected:Nn
\cs_new_protected_nopar:Nn
```

Within `functional` package, there is only one command (`\PrgNewFunction`) for defining new functions, which is good for normal users. The created functions are always protected and accept `\par` in their arguments.

Since `functional` package gets the results of functions by evaluation (including expansion and execution by TEX), it is natural to protect all functions.

1.5 Variants of Arguments

Within `expl3`, there are several expansion variants for arguments, and many expansion functions for expanding them, which are necessary for power users.

```
\module_foo:c
\module_bar:e
\module_bar:x
\module_bar:f
\module_bar:o
\module_bar:V
\module_bar:v
```

```
\exp_args:Nc
\exp_args:Ne
\exp_args:Nx
\exp_args:Nf
\exp_args:No
\exp_args:NV
\exp_args:Nv
```

Within `functional` package, there are only three variants (`c`, `e`, `V`) are provided, and these variants are defined as functions (`\Name`, `\Expand`, `\Value`, respetively), which are easier to use for normal users.

```
\newcommand\test{uvw}
\Name{test}
```

uvw

```
\newcommand\test{uvw}
\Expand{111\test222}
```

111uvw222

```
\IntSet\lTmpaInt{123}
\Value\lTmpaInt
```

123

The most interesting feature is that you can compose these functions. For example, you can easily get the `v` variant of `expl3` by simply composing `\Name` and `\Value` functions:

```
\IntSet\lTmpaInt{123}
\Value{\Name{lTmpaInt}}
```

123

Chapter 2

Basic Definitions (l3basics)

2.1 Defining Functions and Conditionals

\PrgNewFunction ⟨*function*⟩ {⟨*argument specification*⟩} {⟨*code*⟩}

Creates protected ⟨*function*⟩ for evaluating the ⟨*code*⟩. Within the ⟨*code*⟩, the parameters (#1, #2, *etc.*) will be replaced by those absorbed by the function. The returned value *must* be passed with \Result function. The definition is global and an error results if the ⟨*function*⟩ is already defined.

The {⟨*argument specification*⟩} in a list of letters, where each letter is one of the following argument specifiers (nearly all of them are M or m for functions provided by this package):

- M single-token argument, which will be manipulated first
- m multi-token argument, which will be manipulated first
- N single-token argument, which will not be manipulated first
- n multi-token argument, which will not be manipulated first

The argument manipulation for argument type M or m is: if the argument starts with a function defined with \PrgNewFunction, the argument will be evaluated and replaced with the returned value.

\PrgNewConditional ⟨*function*⟩ {⟨*argument specification*⟩} {⟨*code*⟩}

Creates protected conditional ⟨*function*⟩ for evaluating the ⟨*code*⟩. The returned value of the ⟨*function*⟩ *must* be either \cTrueBool or \cFalseBool and be passed with \Result function.. The definition is global and an error results if the ⟨*function*⟩ is already defined.

Assume the ⟨*function*⟩ is \FooIfBar, then another function \FooIfBarTF will be created at the same time. \FooIfBarTF function has two extra arguments which are {⟨*true code*⟩} and {⟨*false code*⟩}.

\Result {⟨*tokens*⟩}

Appends ⟨*tokens*⟩ to \gResultTl, which holds the returned value of current function. This function is normally used in the ⟨*code*⟩ of \PrgNewFunction and \PrgNewConditional.

2.2 Expanding and Using Tokens

\Name {⟨*control sequence name*⟩}

Expands the ⟨*control sequence name*⟩ until only characters remain, then converts this into a control sequence and returns it. The ⟨*control sequence name*⟩ must consist of character tokens when exhaustively expanded.

`\Value` ⟨*variable*⟩

Recovers the content of a ⟨*variable*⟩ and returns the value. An error is raised if the variable does not exist or if it is invalid. Note that it is the same as `\TlUse` for ⟨*tl var*⟩, or `\IntUse` for ⟨*int var*⟩.

`\Expand` {⟨*tokens*⟩}

Expands the ⟨*tokens*⟩ exhaustively and returns the result.

`\ExpNot` {⟨*tokens*⟩}

Prevents expansion of the ⟨*tokens*⟩ inside the argument of `\Expand` function. The argument of `\ExpNot` *must* be surrounded by braces.

`\ExpValue` ⟨*variable*⟩

Recovers the content of the ⟨*variable*⟩, then prevents expansion of this material inside the argument of `\Expand` function.

`\UseOne` {⟨*argument*⟩}
`\GobbleOne` {⟨*argument*⟩}

The function `\UseOne` absorbs one argument and returns it. `\GobbleOne` absorbs one argument and returns nothing. For example

```
\UseOne{abc}\GobbleOne{ijk}\UseOne{xyz}
```
abcxyz

`\UseGobble` {⟨arg_1⟩} {⟨arg_2⟩}
`\GobbleUse` {⟨arg_1⟩} {⟨arg_2⟩}

These functions absorb two arguments. The function `\UseGobble` discards the second argument, and returns the content of the first argument. `\GobbleUse` discards the first argument, and returns the content of the second argument. For example

```
\UseGobble{abc}{uvw}\GobbleUse{abc}{uvw}
```
abcuvw

Chapter 3

Control Structures (l3prg)

3.1 Scratch Variables of Booleans

\lTmpaBool \lTmpbBool \lTmpcBool \lTmpiBool \lTmpjBool \lTmpkBool

Scratch booleans for local assignment. These are never used by the `functional` package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

\gTmpaBool \gTmpbBool \gTmpcBool \gTmpiBool \gTmpjBool \gTmpkBool

Scratch booleans for global assignment. These are never used by the `functional` package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

3.2 Public Functions for Booleans

\BoolNew ⟨*boolean*⟩

Creates a new ⟨*boolean*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*boolean*⟩ is initially `false`.

\BoolSetTrue ⟨*boolean*⟩

Sets ⟨*boolean*⟩ logically `true`.

\BoolSetFalse ⟨*boolean*⟩

Sets ⟨*boolean*⟩ logically `false`.

\BoolIf ⟨*boolean*⟩
\BoolIfTF ⟨*boolean*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests the current truth of ⟨*boolean*⟩, and continues evaluation based on this result. For example

```
\BoolSetTrue\lTmpaBool
\BoolIfTF\lTmpaBool{\Result{True!}}{\Result{False!}}
\BoolSetFalse\lTmpaBool
\BoolIfTF\lTmpaBool{\Result{True!}}{\Result{False!}}
```

True! False!

Chapter 4

Token Lists (`l3tl`)

4.1 Scratch Variables of Token Lists

`\lTmpaTl` `\lTmpbTl` `\lTmpcTl` `\lTmpiTl` `\lTmpjTl` `\lTmpkTl`

Scratch token lists for local assignment. These are never used by the `functional` package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

`\gTmpaTl` `\gTmpbTl` `\gTmpcTl` `\gTmpiTl` `\gTmpjTl` `\gTmpkTl`

Scratch token lists for global assignment. These are never used by the `functional` package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

4.2 Public Functions for Token Lists

`\TlNew` ⟨*tl var*⟩

Creates a new ⟨*tl var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*tl var*⟩ is initially empty.

`\TlUse` ⟨*tl var*⟩

Recovers the content of a ⟨*tl var*⟩ and returns the value. An error is raised if the variable does not exist or if it is invalid. Note that it is possible to use a ⟨*tl var*⟩ directly without an accessor function.

`\TlSet` ⟨*tl var*⟩ `{`⟨*tokens*⟩`}`

Sets ⟨*tl var*⟩ to contain ⟨*tokens*⟩, removing any previous content from the variable. For example

```
\TlSet\lTmpiTl{\IntMathMult{4}{5}}
\TlUse\lTmpiTl
```

20

`\TlClear` ⟨*tl var*⟩

Clears all entries from the ⟨*tl var*⟩. For example

```
\TlSet\lTmpjTl{One}
\TlClear\lTmpjTl
\TlSet\lTmpjTl{Two}
\TlUse\lTmpjTl
```

Two

\TlPutLeft ⟨*tl var*⟩ {⟨*tokens*⟩}

Appends ⟨*tokens*⟩ to the left side of the current content of ⟨*tl var*⟩. For example

```
\TlSet\lTmpkTl{Functional}
\TlPutLeft\lTmpkTl{Hello}
\TlUse\lTmpkTl
```

HelloFunctional

\TlPutRight ⟨*tl var*⟩ {⟨*tokens*⟩}

Appends ⟨*tokens*⟩ to the right side of the current content of ⟨*tl var*⟩. For example

```
\TlSet\lTmpkTl{Functional}
\TlPutRight\lTmpkTl{World}
\TlUse\lTmpkTl
```

FunctionalWorld

\TlIfEmpty ⟨*tl var*⟩
\TlIfEmptyTF ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list variable*⟩ is entirely empty (*i.e.* contains no tokens at all). For example

```
\TlSet\lTmpaTl{abc}
\TlIfEmptyTF\lTmpaTl{\Result{Empty}}{\Result{NonEmpty}}
\TlClear\lTmpaTl
\TlIfEmptyTF\lTmpaTl{\Result{Empty}}{\Result{NonEmpty}}
```

NonEmpty Empty

\TlIfEq ⟨*tl var*$_1$⟩ ⟨*tl var*$_2$⟩
\TlIfEqTF ⟨*tl var*$_1$⟩ ⟨*tl var*$_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Compares the content of two ⟨*token list variables*⟩ and is logically `true` if the two contain the same list of tokens (*i.e.* identical in both the list of characters they contain and the category codes of those characters). For example

```
\TlSet\lTmpaTl{abc}
\TlSet\lTmpbTl{abc}
\TlSet\lTmpcTl{xyz}
\TlIfEqTF\lTmpaTl\lTmpbTl{\Result{Yes}}{\Result{No}}
\TlIfEqTF\lTmpaTl\lTmpcTl{\Result{Yes}}{\Result{No}}
```

Yes No

Chapter 5

Integers (`l3int`)

5.1 Scratch Variables of Integers

`\lTmpaInt` `\lTmpbInt` `\lTmpcInt` `\lTmpiInt` `\lTmpjInt` `\lTmpkInt`

Scratch integer for local assignment. These are never used by the `functional` package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

`\gTmpaInt` `\gTmpbInt` `\gTmpcInt` `\gTmpiInt` `\gTmpjInt` `\gTmpkInt`

Scratch integer for global assignment. These are never used by the `functional` package, and so are safe for use with any function. However, they may be overwritten by other code and so should only be used for short-term storage.

5.2 Public Functions for Integers

`\IntEval` {⟨*integer expression*⟩}

Evaluates the ⟨*integer expression*⟩ and returns the result: for positive results an explicit sequence of decimal digits not starting with `0`, for negative results `-` followed by such a sequence, and `0` for zero. For example

```
\IntEval{(1+4)*(2-3)/5}
```
-1

`\IntMathAdd` {⟨*integer expression*$_1$⟩} {⟨*integer expression*$_2$⟩}

Adds {⟨*integer expression*$_1$⟩} and {⟨*integer expression*$_2$⟩}, and returns the result. For example

```
\IntMathAdd{7}{3}
```
10

`\IntMathSub` {⟨*integer expression*$_1$⟩} {⟨*integer expression*$_2$⟩}

Subtracts {⟨*integer expression*$_1$⟩} from {⟨*integer expression*$_2$⟩}, and returns the result. For example

```
\IntMathSub{7}{3}
```
4

`\IntMathMult` {⟨*integer expression*$_1$⟩} {⟨*integer expression*$_2$⟩}

Multiplies {⟨*integer expression*$_1$⟩} by {⟨*integer expression*$_2$⟩}, and returns the result. For example

```
\IntMathMult{7}{3}
```
21

`\IntMathDiv` {⟨*integer expression*$_1$⟩} {⟨*integer expression*$_2$⟩}

Divides {⟨*integer expression*$_1$⟩} by {⟨*integer expression*$_2$⟩}, and returns the result. For example

```
\IntMathDiv{7}{3}
```
2

`\IntNew` ⟨*integer*⟩

Creates a new ⟨*integer*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*integer*⟩ is initially equal to 0.

`\IntUse` ⟨*integer*⟩

Recovers the content of an ⟨*integer*⟩ and returns the value. An error is raised if the variable does not exist or if it is invalid. Can be omitted in places where an ⟨*integer*⟩ is required (such as in the first and third arguments of `\IntCompareTF`).

`\IntSet` ⟨*integer*⟩ {⟨*integer expression*⟩}

Sets ⟨*integer*⟩ to the value of ⟨*integer expression*⟩, which must evaluate to an integer (as described for `\IntEval`). For example

```
\IntSet\lTmpaInt{3+5}
\IntUse\lTmpaInt
```
8

`\IntZero` ⟨*integer*⟩

Sets ⟨*integer*⟩ to 0. For example

```
\IntSet\lTmpaInt{5}
\IntZero\lTmpaInt
\IntUse\lTmpaInt
```
0

`\IntIncr` ⟨*integer*⟩

Increases the value stored in ⟨*integer*⟩ by 1. For example

```
\IntSet\lTmpaInt{5}
\IntIncr\lTmpaInt
\IntUse\lTmpaInt
```
6

`\IntDecr` ⟨*integer*⟩

Decreases the value stored in ⟨*integer*⟩ by 1. For example

```
\IntSet\lTmpaInt{5}
\IntDecr\lTmpaInt
\IntUse\lTmpaInt
```
4

\IntAdd ⟨*integer*⟩ {⟨*integer expression*⟩}

Adds the result of the ⟨*integer expression*⟩ to the current content of the ⟨*integer*⟩. For example

```
\IntSet\lTmpaInt{5}
\IntAdd\lTmpaInt{2}
\IntUse\lTmpaInt
```
7

\IntSub ⟨*integer*⟩ {⟨*integer expression*⟩}

Subtracts the result of the ⟨*integer expression*⟩ from the current content of the ⟨*integer*⟩. For example

```
\IntSet\lTmpaInt{5}
\IntSub\lTmpaInt{3}
\IntUse\lTmpaInt
```
2

\IntStepVariable {⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} ⟨*tl var*⟩ {⟨*code*⟩}

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be integer expressions. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is evaluated, with the ⟨*tl var*⟩ defined as the current ⟨*value*⟩. Thus the ⟨*code*⟩ should make use of the ⟨*tl var*⟩. For example

```
\TlClear\lTmpaTl
\IntStepVariable{1}{3}{30}\lTmpiTl{
  \TlPutRight\lTmpaTl{\Value\lTmpiTl}
  \TlPutRight\lTmpaTl{ }
}
\Result{\Value\lTmpaTl}
```
1 4 7 10 13 16 19 22 25 28

\IntCompare {⟨*intexpr*$_1$⟩} ⟨*relation*⟩ {⟨*intexpr*$_2$⟩}
\IntCompareTF {⟨*intexpr*$_1$⟩} ⟨*relation*⟩ {⟨*intexpr*$_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}

This function first evaluates each of the ⟨*integer expressions*⟩ as described for \IntEval. The two results are then compared using the ⟨*relation*⟩:

Equal	=
Greater than	>
Less than	<

For example

```
\IntCompareTF{2}>{1}{\Result{Greater}}{\Result{Less}}
\IntCompareTF{2}>{3}{\Result{Greater}}{\Result{Less}}
```
Greater Less

Chapter 6

The Source Code

```
%% ------------------------------------------------------------------------
%% Functional: LaTeX2 functional interfaces to LaTeX3 programming layer
%% Copyright : 2022 (c) Jianrui Lyu <tolvjr@163.com>
%% Repository: https://github.com/lvjr/functional
%% License   : The LaTeX Project Public License 1.3c
%% ------------------------------------------------------------------------
```

6.1 Interfaces for Function Definitions (l3basics)

```
\NeedsTeXFormat{LaTeX2e}[2018-04-01]

\RequirePackage{expl3}
\ProvidesExplPackage{functional}{2022-03-10}{2022A}
  {^^JLaTeX2 functional interfaces to LaTeX3 programming layer}

\cs_generate_variant:Nn \iow_log:n { V }
\cs_generate_variant:Nn \tl_set:Nn { Ne }

\tl_new:N \gResultTl
\int_new:N \l__fun_arg_count_int
\tl_new:N \l__fun_parameters_defined_tl
\tl_const:Nn \c__fun_parameter_defined_i__tl     {  } % no argument
\tl_const:Nn \c__fun_parameter_defined_i_i_tl    { #1 }
\tl_const:Nn \c__fun_parameter_defined_i_ii_tl   { #1 #2 }
\tl_const:Nn \c__fun_parameter_defined_i_iii_tl  { #1 #2 #3 }
\tl_const:Nn \c__fun_parameter_defined_i_iv_tl   { #1 #2 #3 #4 }
\tl_const:Nn \c__fun_parameter_defined_i_v_tl    { #1 #2 #3 #4 #5 }
\tl_const:Nn \c__fun_parameter_defined_i_vi_tl   { #1 #2 #3 #4 #5 #6 }
\tl_const:Nn \c__fun_parameter_defined_i_vii_tl  { #1 #2 #3 #4 #5 #6 #7 }
\tl_const:Nn \c__fun_parameter_defined_i_viii_tl { #1 #2 #3 #4 #5 #6 #7 #8 }
\tl_const:Nn \c__fun_parameter_defined_i_ix_tl   { #1 #2 #3 #4 #5 #6 #7 #8 #9 }
\tl_new:N \l__fun_parameters_called_tl
\tl_const:Nn \c__fun_parameter_called_i_i_tl   { {#1} }
\tl_const:Nn \c__fun_parameter_called_i_ii_tl  { {#1}{#2} }
\tl_const:Nn \c__fun_parameter_called_i_iii_tl { {#1}{#2}{#3} }
\tl_const:Nn \c__fun_parameter_called_i_iv_tl  { {#1}{#2}{#3}{#4} }
\tl_const:Nn \c__fun_parameter_called_i_v_tl   { {#1}{#2}{#3}{#4}{#5} }
\tl_const:Nn \c__fun_parameter_called_i_vi_tl  { {#1}{#2}{#3}{#4}{#5}{#6} }
\tl_const:Nn \c__fun_parameter_called_i_vii_tl { {#1}{#2}{#3}{#4}{#5}{#6}{#7} }
\tl_new:N \l__fun_parameters_true_tl
\tl_new:N \l__fun_parameters_false_tl
```

```
\tl_const:Nn \c__fun_parameter_called_i_tl    { {#1} }
\tl_const:Nn \c__fun_parameter_called_ii_tl   { {#2} }
\tl_const:Nn \c__fun_parameter_called_iii_tl  { {#3} }
\tl_const:Nn \c__fun_parameter_called_iv_tl   { {#4} }
\tl_const:Nn \c__fun_parameter_called_v_tl    { {#5} }
\tl_const:Nn \c__fun_parameter_called_vi_tl   { {#6} }
\tl_const:Nn \c__fun_parameter_called_vii_tl  { {#7} }
\tl_const:Nn \c__fun_parameter_called_viii_tl { {#8} }
\tl_const:Nn \c__fun_parameter_called_ix_tl   { {#9} }
\tl_new:N \l__fun_argument_tl
\tl_new:N \l__fun_argument_i_tl
\tl_new:N \l__fun_argument_ii_tl
\tl_new:N \l__fun_argument_iii_tl
\tl_new:N \l__fun_argument_iv_tl
\tl_new:N \l__fun_argument_v_tl
\tl_new:N \l__fun_argument_vi_tl
\tl_new:N \l__fun_argument_vii_tl
\tl_new:N \l__fun_argument_viii_tl
\tl_new:N \l__fun_argument_ix_tl

%% #1: function name; #2: argument specification; #3 function body
\cs_new_protected:Npn \__fun_new_function:Nnn #1 #2 #3
  {
    \int_set:Nn \l__fun_arg_count_int { \tl_count:n {#2} } % spaces are ignored
    \tl_set_eq:Nc \l__fun_parameters_defined_tl
      { c__fun_parameter_defined_i_ \int_to_roman:n { \l__fun_arg_count_int } _tl }
    \exp_last_unbraced:NcV \cs_new_protected:Npn
      { __fun_defined_ \cs_to_str:N #1 : w }
      \l__fun_parameters_defined_tl
      {
        \__fun_group_begin:
        \tl_gclear:N \gResultTl
        #3
        \__fun_tracing_log:e { [0] ~ \gResultTl }
        \__fun_group_end:
      }
    \use:c { __fun_new_with_arg_ \int_to_roman:n { \l__fun_arg_count_int } :NnV }
      #1 {#2} \l__fun_parameters_defined_tl
  }
\cs_generate_variant:Nn \__fun_new_function:Nnn { cne }

\cs_set_eq:NN \PrgNewFunction \__fun_new_function:Nnn

\tl_new:N \g__fun_last_result_tl

%% #1: function name; #2: argument specification; #3 function body
\cs_new_protected:Npn \__fun_new_conditional:Nnn #1 #2 #3
  {
    \__fun_new_function:Nnn #1 { #2 } { #3 }
    \tl_set_eq:Nc \l__fun_parameters_called_tl
      { c__fun_parameter_called_i_ \int_to_roman:n { \l__fun_arg_count_int } _tl }
    \tl_set_eq:Nc \l__fun_parameters_true_tl
      { c__fun_parameter_called_ \int_to_roman:n { \l__fun_arg_count_int + 1 } _tl }
    \tl_set_eq:Nc \l__fun_parameters_false_tl
      { c__fun_parameter_called_ \int_to_roman:n { \l__fun_arg_count_int + 2 } _tl }
    \__fun_new_function:cne { \cs_to_str:N #1 TF } { #2 n n }
      {
```

```
        #1 \exp_not:V \l__fun_parameters_called_tl
        \exp_not:n
          {
            \tl_set_eq:NN \g__fun_last_result_tl \gResultTl
            \tl_gclear:N \gResultTl
            \exp_last_unbraced:NV \bool_if:NTF \g__fun_last_result_tl
          }
        \exp_not:V \l__fun_parameters_true_tl
        \exp_not:V \l__fun_parameters_false_tl
      }
  }

\cs_set_eq:NN \PrgNewConditional \__fun_new_conditional:Nnn

%% #1: function name; #2: argument specifications; #3 parameters tl defined
%% Some times we need to create a function without arguments
\cs_new_protected:Npn \__fun_new_with_arg_:Nnn #1 #2 #3
  {
    \cs_new_protected:Npn #1 #3
      {
        \__fun_evaluate:Nn #1 {#2}
      }
  }
\cs_generate_variant:Nn \__fun_new_with_arg_:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_i:Nnn #1 #2 #3
  {
    \cs_new_protected:Npn #1 #3
      {
        \tl_set:Nn \l__fun_argument_i_tl { ##1 }
        \__fun_evaluate:Nn #1 {#2}
      }
  }
\cs_generate_variant:Nn \__fun_new_with_arg_i:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_ii:Nnn #1 #2 #3
  {
    \cs_new_protected:Npn #1 #3
      {
        \tl_set:Nn \l__fun_argument_i_tl  { ##1 }
        \tl_set:Nn \l__fun_argument_ii_tl { ##2 }
        \__fun_evaluate:Nn #1 {#2}
      }
  }
\cs_generate_variant:Nn \__fun_new_with_arg_ii:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_iii:Nnn #1 #2 #3
  {
    \cs_new_protected:Npn #1 #3
      {
        \tl_set:Nn \l__fun_argument_i_tl   { ##1 }
        \tl_set:Nn \l__fun_argument_ii_tl  { ##2 }
        \tl_set:Nn \l__fun_argument_iii_tl { ##3 }
        \__fun_evaluate:Nn #1 {#2}
```

```
      }
  }
\cs_generate_variant:Nn \__fun_new_with_arg_iii:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_iv:Nnn #1 #2 #3
  {
    \cs_new_protected:Npn #1 #3
      {
        \tl_set:Nn \l__fun_argument_i_tl   { ##1 }
        \tl_set:Nn \l__fun_argument_ii_tl  { ##2 }
        \tl_set:Nn \l__fun_argument_iii_tl { ##3 }
        \tl_set:Nn \l__fun_argument_iv_tl  { ##4 }
        \__fun_evaluate:Nn #1 {#2}
      }
  }
\cs_generate_variant:Nn \__fun_new_with_arg_iv:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_v:Nnn #1 #2 #3
  {
    \cs_new_protected:Npn #1 #3
      {
        \tl_set:Nn \l__fun_argument_i_tl   { ##1 }
        \tl_set:Nn \l__fun_argument_ii_tl  { ##2 }
        \tl_set:Nn \l__fun_argument_iii_tl { ##3 }
        \tl_set:Nn \l__fun_argument_iv_tl  { ##4 }
        \tl_set:Nn \l__fun_argument_v_tl   { ##5 }
        \__fun_evaluate:Nn #1 {#2}
      }
  }
\cs_generate_variant:Nn \__fun_new_with_arg_v:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_vi:Nnn #1 #2 #3
  {
    \cs_new_protected:Npn #1 #3
      {
        \tl_set:Nn \l__fun_argument_i_tl   { ##1 }
        \tl_set:Nn \l__fun_argument_ii_tl  { ##2 }
        \tl_set:Nn \l__fun_argument_iii_tl { ##3 }
        \tl_set:Nn \l__fun_argument_iv_tl  { ##4 }
        \tl_set:Nn \l__fun_argument_v_tl   { ##5 }
        \tl_set:Nn \l__fun_argument_vi_tl  { ##6 }
        \__fun_evaluate:Nn #1 {#2}
      }
  }
\cs_generate_variant:Nn \__fun_new_with_arg_vi:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_vii:Nnn #1 #2 #3
  {
    \cs_new_protected:Npn #1 #3
      {
        \tl_set:Nn \l__fun_argument_i_tl   { ##1 }
        \tl_set:Nn \l__fun_argument_ii_tl  { ##2 }
        \tl_set:Nn \l__fun_argument_iii_tl { ##3 }
```

```
        \tl_set:Nn \l__fun_argument_iv_tl  { ##4 }
        \tl_set:Nn \l__fun_argument_v_tl   { ##5 }
        \tl_set:Nn \l__fun_argument_vi_tl  { ##6 }
        \tl_set:Nn \l__fun_argument_vii_tl { ##7 }
        \__fun_evaluate:Nn #1 {#2}
      }
  }
\cs_generate_variant:Nn \__fun_new_with_arg_vii:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_viii:Nnn #1 #2 #3
  {
    \cs_new_protected:Npn #1 #3
      {
        \tl_set:Nn \l__fun_argument_i_tl    { ##1 }
        \tl_set:Nn \l__fun_argument_ii_tl   { ##2 }
        \tl_set:Nn \l__fun_argument_iii_tl  { ##3 }
        \tl_set:Nn \l__fun_argument_iv_tl   { ##4 }
        \tl_set:Nn \l__fun_argument_v_tl    { ##5 }
        \tl_set:Nn \l__fun_argument_vi_tl   { ##6 }
        \tl_set:Nn \l__fun_argument_vii_tl  { ##7 }
        \tl_set:Nn \l__fun_argument_viii_tl { ##8 }
        \__fun_evaluate:Nn #1 {#2}
      }
  }
\cs_generate_variant:Nn \__fun_new_with_arg_viii:Nnn { NnV }

%% #1: function name; #2: argument specifications; #3 parameters tl defined
\cs_new_protected:Npn \__fun_new_with_arg_ix:Nnn #1 #2 #3
  {
    \cs_new_protected:Npn #1 #3
      {
        \tl_set:Nn \l__fun_argument_i_tl    { ##1 }
        \tl_set:Nn \l__fun_argument_ii_tl   { ##2 }
        \tl_set:Nn \l__fun_argument_iii_tl  { ##3 }
        \tl_set:Nn \l__fun_argument_iv_tl   { ##4 }
        \tl_set:Nn \l__fun_argument_v_tl    { ##5 }
        \tl_set:Nn \l__fun_argument_vi_tl   { ##6 }
        \tl_set:Nn \l__fun_argument_vii_tl  { ##7 }
        \tl_set:Nn \l__fun_argument_viii_tl { ##8 }
        \tl_set:Nn \l__fun_argument_ix_tl   { ##9 }
        \__fun_evaluate:Nn #1 {#2}
      }
  }
\cs_generate_variant:Nn \__fun_new_with_arg_ix:Nnn { NnV }

\int_new:N \g__fun_nesting_level_int
\int_new:N \l__fun_argtype_number_int
\tl_new:N \l__fun_argtype_tl
\tl_const:Nn \c__fun_argtype_m_tl { m }
\tl_const:Nn \c__fun_argtype_M_tl { M }
\tl_const:Nn \c__fun_argtype_n_tl { n }
\tl_const:Nn \c__fun_argtype_N_tl { N }

%% #1: function name; #2: argument specifications
\cs_new_protected:Npn \__fun_evaluate:Nn #1 #2
  {
```

```
    \int_zero:N \l__fun_argtype_number_int
    \int_gincr:N \g__fun_nesting_level_int
    \__fun_arguments_gclear:
    \tl_map_variable:nNn { #2 } \l__fun_argtype_tl % spaces are ignored
      {
        \int_incr:N \l__fun_argtype_number_int
        \tl_set_eq:Nc \l__fun_argument_tl
          { l__fun_argument_ \int_to_roman:n { \l__fun_argtype_number_int } _tl }
        \tl_case:Nn \l__fun_argtype_tl
          {
            \c__fun_argtype_m_tl
              {
                \__fun_evaluate_and_put_argument:N \l__fun_argument_tl
              }
            \c__fun_argtype_M_tl
              {
                \__fun_evaluate_and_put_argument:N \l__fun_argument_tl
              }
            \c__fun_argtype_n_tl
              {
                \__fun_arguments_gput:e { { \exp_not:V \l__fun_argument_tl } }
              }
            \c__fun_argtype_N_tl
              {
                \__fun_arguments_gput:e { \exp_not:V \l__fun_argument_tl }
              }
          }
      }
    \__fun_arguments_log:N #1
    \__fun_arguments_called:c { __fun_defined_ \cs_to_str:N #1 : w }
    \int_gdecr:N \g__fun_nesting_level_int
    \__fun_return_result:
  }

\cs_set_eq:NN \__fun_cs_temp:w \scan_stop:

\cs_new_protected:Npn \__fun_evaluate_and_put_argument:N #1
  {
    \cs_set_eq:Nc \__fun_cs_temp:w
      {
        __fun_defined_ \exp_last_unbraced:Ne \cs_to_str:N { \tl_head:N #1 } : w
      }
    \cs_if_exist:NTF \__fun_cs_temp:w
      {
        #1
        \__fun_arguments_gput:e { { \exp_not:V \gResultTl } }
      }
      {
        \__fun_arguments_gput:e { { \exp_not:V #1 } }
      }
  }

\cs_new_protected:Npn \__fun_arguments_called:N #1
  {
    \exp_last_unbraced:Nv
      #1 { g__fun_arguments_ \int_use:N \g__fun_nesting_level_int _tl }
  }
\cs_generate_variant:Nn \__fun_arguments_called:N { c }
```

```
\cs_new_protected:Npn \__fun_arguments_gclear:
  {
    \tl_gclear:c { g__fun_arguments_ \int_use:N \g__fun_nesting_level_int _tl }
  }

\cs_new_protected:Npn \__fun_arguments_log:N #1
  {
    \__fun_tracing_log:e
    {
      [I] ~ \token_to_str:N #1
     \exp_not:v { g__fun_arguments_ \int_use:N \g__fun_nesting_level_int _tl }
    }
  }

\cs_new_protected:Npn \__fun_arguments_gput:n #1
  {
    \tl_gput_right:cn
      { g__fun_arguments_ \int_use:N \g__fun_nesting_level_int _tl } { #1 }
  }
\cs_generate_variant:Nn \__fun_arguments_gput:n { e }

\cs_new_protected:Npn \__fun_put_result:n #1
  {
    \tl_gput_right:Nn \gResultTl { #1 }
  }
\cs_generate_variant:Nn \__fun_put_result:n { e, V }

\PrgNewFunction \Result { m }
  {
    \__fun_put_result:n { #1 }
  }

\PrgNewFunction \Name { m }
  {
    \exp_args:Nc \__fun_put_result:n { #1 }
  }
\cs_set_eq:NN \UseName \Name

\PrgNewFunction \Expand { m }
  {
    \__fun_put_result:e { #1 }
  }
\cs_set_eq:NN \UseExpand \Expand

\PrgNewFunction \Value { M }
  {
    \__fun_put_result:V #1
  }
\cs_set_eq:NN \UseValue \Value

\cs_set_eq:NN \ExpNot \exp_not:n
\cs_set_eq:NN \ExpValue \exp_not:V

\cs_new_protected:Npn \__fun_return_result:
  {
    \int_compare:nNnT { \g__fun_nesting_level_int } = { 0 }
```

```
      { \tl_use:N \gResultTl }
  }

\tl_new:N \l__fun_variable_type_tl

\prg_new_protected_conditional:Npnn \__fun_if_global_variable:N #1 { TF }
  {
    \tl_set:Ne \l__fun_variable_type_tl
      { \exp_args:Ne \tl_head:n { \cs_to_str:N #1 } }
    \str_if_eq:VnTF \l__fun_variable_type_tl { g }
      { \prg_return_true: }
      {
        \str_if_eq:VnTF \l__fun_variable_type_tl { c }
          { \prg_return_true: }
          { \prg_return_false: }
      }
  }

%% We must not put an assignment inside a group
\cs_new_protected:Npn \__fun_do_assignment:Nnn #1 #2 #3
  {
    \__fun_group_end:
    \__fun_if_global_variable:NTF #1 { #2 } { #3 }
    \__fun_group_begin:
  }

\bool_new:N \l__fun_scoping_bool

\cs_new_protected:Npn \__fun_scoping_true:
  {
    \cs_set_eq:NN \__fun_group_begin: \group_begin:
    \cs_set_eq:NN \__fun_group_end: \group_end:
  }

\cs_new_protected:Npn \__fun_scoping_false:
  {
    \cs_set_eq:NN \__fun_group_begin: \scan_stop:
    \cs_set_eq:NN \__fun_group_end: \scan_stop:
  }

\cs_new_protected:Npn \__fun_scoping_set:
  {
    \bool_if:NTF \l__fun_scoping_bool
      { \__fun_scoping_true: } { \__fun_scoping_false: }
  }

\bool_new:N \l__fun_tracing_bool
\tl_new:N \l__tracing_text_tl

\cs_new_protected:Npn \__fun_tracing_log_on:n #1
  {
    \tl_set:Ne \l__tracing_text_tl
      {
        \prg_replicate:nn
          { \int_eval:n { (\g__fun_nesting_level_int - 1) * 4 } } { ~ }
      }
```

```
    \tl_put_right:Nn \l__tracing_text_tl { #1 }
    \iow_log:V \l__tracing_text_tl
  }
\cs_generate_variant:Nn \__fun_tracing_log_on:n { e, V }

\cs_new_protected:Npn \__fun_tracing_log_off:n #1 { }
\cs_generate_variant:Nn \__fun_tracing_log_off:n { e, V }

\cs_new_protected:Npn \__fun_tracing_true:
  {
    \cs_set_eq:NN \__fun_tracing_log:n \__fun_tracing_log_on:n
    \cs_set_eq:NN \__fun_tracing_log:e \__fun_tracing_log_on:e
    \cs_set_eq:NN \__fun_tracing_log:V \__fun_tracing_log_on:V
  }

\cs_new_protected:Npn \__fun_tracing_false:
  {
    \cs_set_eq:NN \__fun_tracing_log:n \__fun_tracing_log_off:n
    \cs_set_eq:NN \__fun_tracing_log:e \__fun_tracing_log_off:e
    \cs_set_eq:NN \__fun_tracing_log:V \__fun_tracing_log_off:V
  }

\cs_new_protected:Npn \__fun_tracing_set:
  {
    \bool_if:NTF \l__fun_tracing_bool
      { \__fun_tracing_true: } { \__fun_tracing_false: }
  }

\keys_define:nn { functional }
  {
    scoping .bool_set:N = \l__fun_scoping_bool,
    tracing .bool_set:N = \l__fun_tracing_bool,
  }

\NewDocumentCommand \Functional { m }
  {
    \keys_set:nn { functional } { #1 }
    \__fun_scoping_set:
    \__fun_tracing_set:
  }

\Functional { scoping = false, tracing = false }

\PrgNewFunction \UseOne { n }
  {
    \Result { #1 }
  }

\PrgNewFunction \GobbleOne { n }
  {
    \Result { }
  }

\PrgNewFunction \UseGobble { n n }
  {
```

```
    \UseOne { #1 }
  }

\PrgNewFunction \GobbleUse { n n }
  {
    \UseOne { #2 }
  }
```

6.2 Interfaces for Control Structures (l3prg)

```
\bool_const:Nn \cTrueBool { \c_true_bool }
\bool_const:Nn \cFalseBool { \c_false_bool }

\bool_new:N \lTmpaBool   \bool_new:N \lTmpbBool   \bool_new:N \lTmpcBool
\bool_new:N \lTmpiBool   \bool_new:N \lTmpjBool   \bool_new:N \lTmpkBool
\bool_new:N \l@Funx@Bool \bool_new:N \l@Funy@Bool \bool_new:N \l@Funz@Bool

\bool_new:N \gTmpaBool   \bool_new:N \gTmpbBool   \bool_new:N \gTmpcBool
\bool_new:N \gTmpiBool   \bool_new:N \gTmpjBool   \bool_new:N \gTmpkBool
\bool_new:N \g@Funx@Bool \bool_new:N \g@Funy@Bool \bool_new:N \g@Funz@Bool

\cs_set_eq:NN \BoolNew \bool_new:N
\cs_set_eq:NN \BoolLog \bool_log:N

\PrgNewFunction \BoolSetTrue { M }
  {
    \__fun_do_assignment:Nnn #1 { \bool_gset_true:N #1 } { \bool_set_true:N #1 }
  }

\PrgNewFunction \BoolSetFalse { M }
  {
    \__fun_do_assignment:Nnn #1 { \bool_gset_false:N #1 } { \bool_set_false:N #1 }
  }

\PrgNewConditional \BoolIf { N }
  {
    \Result { #1 }
  }
```

6.3 Interfaces for Token Lists (l3tl)

```
\tl_new:N \lTmpaTl   \tl_new:N \lTmpbTl   \tl_new:N \lTmpcTl
\tl_new:N \lTmpiTl   \tl_new:N \lTmpjTl   \tl_new:N \lTmpkTl
\tl_new:N \l@Funx@Tl \tl_new:N \l@Funy@Tl \tl_new:N \l@Funz@Tl

\tl_new:N \gTmpaTl   \tl_new:N \gTmpbTl   \tl_new:N \gTmpcTl
\tl_new:N \gTmpiTl   \tl_new:N \gTmpjTl   \tl_new:N \gTmpkTl
\tl_new:N \g@Funx@Tl \tl_new:N \g@Funy@Tl \tl_new:N \g@Funz@Tl

\cs_set_eq:NN \TlNew \tl_new:N
\cs_set_eq:NN \TlLog \tl_log:N

\PrgNewFunction \TlClear { M }
```

```
  {
    \__fun_do_assignment:Nnn #1 { \tl_gclear:N #1 } { \tl_clear:N #1 }
  }

\PrgNewFunction \TlSet { M m }
  {
    \__fun_do_assignment:Nnn #1 { \tl_gset:Nn #1 {#2} } { \tl_set:Nn #1 {#2} }
  }

\PrgNewFunction \TlPutLeft { M m }
  {
    \__fun_do_assignment:Nnn #1
      { \tl_gput_left:Nn #1 {#2} } { \tl_put_left:Nn #1 {#2} }
  }

\PrgNewFunction \TlPutRight { M m }
  {
    \__fun_do_assignment:Nnn #1
      { \tl_gput_right:Nn #1 {#2} } { \tl_put_right:Nn #1 {#2} }
  }

\PrgNewFunction \TlUse { M }
  {
    \Result { \Value #1 }
  }

\PrgNewConditional \TlIfEmpty { N }
  {
    \tl_if_empty:NTF #1
      { \Result { \cTrueBool } }
      { \Result { \cFalseBool } }
  }

\PrgNewConditional \TlIfEq { N N }
  {
    \tl_if_eq:NNTF #1 #2
      { \Result { \cTrueBool } }
      { \Result { \cFalseBool } }
  }
```

6.4 Interfaces for Integers (l3int)

```
\int_new:N \lTmpaInt   \int_new:N \lTmpbInt   \int_new:N \lTmpcInt
\int_new:N \lTmpiInt   \int_new:N \lTmpjInt   \int_new:N \lTmpkInt
\int_new:N \l@Funx@Int \int_new:N \l@Funy@Int \int_new:N \l@Funz@Int

\int_new:N \gTmpaInt   \int_new:N \gTmpbInt   \int_new:N \gTmpcInt
\int_new:N \gTmpiInt   \int_new:N \gTmpjInt   \int_new:N \gTmpkInt
\int_new:N \g@Funx@Int \int_new:N \g@Funy@Int \int_new:N \g@Funz@Int

\cs_set_eq:NN \IntNew \int_new:N
\cs_set_eq:NN \IntLog \int_log:N

\PrgNewFunction \IntZero { M }
```

```
  {
    \__fun_do_assignment:Nnn #1 { \int_gzero:N #1 } { \int_zero:N #1 }
  }

\PrgNewFunction \IntIncr { M }
  {
    \__fun_do_assignment:Nnn #1 { \int_gincr:N #1 } { \int_incr:N #1 }
  }

\PrgNewFunction \IntDecr { M }
  {
    \__fun_do_assignment:Nnn #1 { \int_gdecr:N #1 } { \int_decr:N #1 }
  }

\PrgNewFunction \IntSet { M m }
  {
    \__fun_do_assignment:Nnn #1 { \int_gset:Nn #1 {#2} } { \int_set:Nn #1 {#2} }
  }

\PrgNewFunction \IntAdd { M m }
  {
    \__fun_do_assignment:Nnn #1 { \int_gadd:Nn #1 {#2} } { \int_add:Nn #1 {#2} }
  }

\PrgNewFunction \IntSub { M m }
  {
    \__fun_do_assignment:Nnn #1 { \int_gsub:Nn #1 {#2} } { \int_sub:Nn #1 {#2} }
  }

\PrgNewFunction \IntUse { M }
  {
    \Result { \Value #1 }
  }

\PrgNewFunction \IntEval { m }
  {
    \Result { \Expand { \int_eval:n { #1 } } }
  }

\PrgNewFunction \IntMathAdd { m m }
  {
    \int_set:Nn \l@Funx@Int { \int_eval:n { #1 + #2 } }
    \Result { \Value \l@Funx@Int }
  }

\PrgNewFunction \IntMathSub { m m }
  {
    \int_set:Nn \l@Funx@Int { \int_eval:n { #1 - #2 } }
    \Result { \Value \l@Funx@Int }
  }

\PrgNewFunction \IntMathMult { m m }
  {
    \int_set:Nn \l@Funx@Int { \int_eval:n { #1 * #2 } }
    \Result { \Value \l@Funx@Int }
```

```
  }

\PrgNewFunction \IntMathDiv { m m }
  {
    \int_set:Nn \l@Funx@Int { \int_eval:n { #1 / #2 } }
    \Result { \Value \l@Funx@Int }
  }

\PrgNewFunction \IntStepVariable { m m m M n }
  {
    \int_step_variable:nnnNn { #1 } { #2 } { #3 } #4 { #5 }
  }

\PrgNewConditional \IntCompare { m N m }
  {
    \int_compare:nNnTF {#1} #2 {#3}
      { \Result { \cTrueBool } }
      { \Result { \cFalseBool } }
  }
```