LaTeX2 Functional Interfaces for LaTeX3 Programming Layer

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

Version 2022B (2022-03-19)

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

1 Overview of Features **5**

1.1 Evaluation from Inside to Outside 5
1.2 Group Scoping of Functions 5
1.3 Tracing Evaluation of Functions 6
1.4 Definitions of Functions 7
1.5 Variants of Arguments 7

2 Functional Progarmming (`Prg`) **9**

2.1 Defining Functions and Conditionals 9
2.2 Collecting Returned Values 9

3 Argument Using (`Use`) **10**

3.1 Expanding Tokens 10
3.2 Using Tokens 10

4 Control Structures (`Bool`) **12**

4.1 Constant and Scratch Booleans 12
4.2 Creating and Setting Booleans 12
4.3 Viewing Booleans 13
4.4 Booleans and Conditionals 13

5 Token Lists (`Tl`) **15**

5.1 Constant and Scratch Token Lists 15
5.2 Creating and Using Token Lists 15
5.3 Viewing Token Lists 16
5.4 Setting Token List Variables 16
5.5 Replacing Tokens 17
5.6 Working with the Content of Token Lists 18
5.7 Mapping over Token Lists 19
5.8 Token List Conditionals 20
5.9 Token List Case Functions 21

6 Strings (`Str`) **23**

6.1 Constant and Scratch Strings 23
6.2 Creating and Using Strings 23
6.3 Viewing Strings 24
6.4 Setting String Variables 24

6.5 Modifying String Variables 25
6.6 Working with the Content of Strings 26
6.7 Mapping over Strings 27
6.8 String Conditionals 27
6.9 String Case Functions 28

7 Integers (`Int`) 30
7.1 Constant and Scratch Integers 30
7.2 Integer Expressions 30
7.3 Creating and Using Integers 32
7.4 Viewing Integers 32
7.5 Setting Integer Variables 33
7.6 Integer Step Functions 34
7.7 Integer Conditionals 34
7.8 Integer Case Functions 35

8 Floating Point Numbers (`Fp`) 37
8.1 Constant and Scratch Floating Points 37
8.2 Floating Point Expressions 38
8.3 Creating and Using Floating Points 39
8.4 Viewing Floating Points 39
8.5 Setting Floating Point Variables 40
8.6 Floating Point Step Functions 40
8.7 Float Point Conditionals 41

9 Dimensions (`Dim`) 42
9.1 Constant and Scratch Dimensions 42
9.2 Dimension Expressions 42
9.3 Creating and Using Dimensions 43
9.4 Viewing Dimensions 44
9.5 Setting Dimension Variables 44
9.6 Dimension Step Functions 45
9.7 Dimension Conditionals 46
9.8 Dimension Case Functions 47

10 Comma Separated Lists (`Clist`) 49
10.1 Constant and Scratch Comma Lists 49
10.2 Creating and Using Comma Lists 49
10.3 Viewing Comma Lists 50
10.4 Setting Comma Lists 50
10.5 Modifying Comma Lists 51
10.6 Working with the Contents of Comma Lists 52
10.7 Comma Lists as Stacks 52
10.8 Mapping over Comma Lists 53
10.9 Comma List Conditionals 53

11 The Source Code 55
11.1 Interfaces for Functional Programming (Prg) 55
11.2 Interfaces for Argument Using (Use) 64
11.3 Interfaces for Control Structures (Bool) 65
11.4 Interfaces for Token Lists (Tl) 66
11.5 Interfaces for Strings (Str) 70
11.6 Interfaces for Integers (Int) 73
11.7 Interfaces for Floating Point Numbers (Fp) 76
11.8 Interfaces for Dimensions (Dim) 79
11.9 Interfaces for Sorting Functions (Sort) 81
11.10Interfaces for Comma Separated Lists (Clist) 81

Chapter 1

Overview of Features

1.1 Evaluation from Inside to Outside

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

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

```
-- define a function --
function MathSquare (arg)
  local lTmpaInt = arg * arg
  return lTmpaInt
end
-- use the function --
print(MathSquare(5))
print(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 `\IgnoreSpacesOn` and `\IgnoreSpacesOff` 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:

```
\IgnoreSpacesOn
\PrgNewFunction \Fact { m } {
  \IntCompareTF {#1} = {0} {
    \Result {1}
  }{
    \Result {\IntMathMult{#1}{\Fact{\IntMathSub{#1}{1}}}}
  }
}
\IgnoreSpacesOff
\Fact{4}
```

```
-- define a function --
function Fact (n)
  if n == 0 then
    return 1
  else
    return n * Fact(n-1)
  end
end
-- use the function --
print(Fact(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. 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}
\IgnoreSpacesOn
\IntSet \lTmpaInt {1}
\IntLogVar \lTmpaInt              % ---- 1
\PrgNewFunction \SomeFun { } {
  \IntSet \lTmpaInt {2}
  \IntLogVar \lTmpaInt            % ---- 2
  \IntCompareTF {1} > {0} {
    \IntSet \lTmpaInt {3}
    \IntLogVar \lTmpaInt          % ---- 3
  }{ }
  \IntLogVar \lTmpaInt            % ---- 2
}
\SomeFun
\IntLogVar \lTmpaInt              % ---- 1
\IgnoreSpacesOff
```

```
-- lua code --
-- begin example --
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
-- end example --
```

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}
\IgnoreSpacesOn
\IntSet \gTmpaInt {1}
\IntLogVar \gTmpaInt              % ---- 1
\PrgNewFunction \SomeFun { } {
  \IntSet \gTmpaInt {2}
  \IntLogVar \gTmpaInt            % ---- 2
  \IntCompareTF {1} > {0} {
    \IntSet \gTmpaInt {3}
    \IntLogVar \gTmpaInt          % ---- 3
  }{ }
  \IntLogVar \gTmpaInt            % ---- 3
}
\SomeFun
\IntLogVar \gTmpaInt              % ---- 3
\IgnoreSpacesOff
```

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

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}}
                [O] 25
            [I] \Result{25}
            [O] 25
        [O] 25
    [I] \IntSet{\lTmpaInt }{25}
    [O]
        [I] \Value{\lTmpaInt }
        [O] 25
    [I] \Result{25}
    [O] 25
[O] 25
[I] \MathSquare{25}
        [I] \IntEval{25*25}
                [I] \Expand{\int_eval:n {25*25}}
                [O] 625
            [I] \Result{625}
            [O] 625
        [O] 625
    [I] \IntSet{\lTmpaInt }{625}
    [O]
        [I] \Value{\lTmpaInt }
        [O] 625
    [I] \Result{625}
    [O] 625
[O] 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:Nn
\cs_new_nopar:Npn                    \cs_new_nopar:Nn
\cs_new_protected:Npn                \cs_new_protected:Nn
\cs_new_protected_nopar:Npn          \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                        \exp_args:Nc
\module_bar:e                        \exp_args:Ne
\module_bar:x                        \exp_args:Nx
\module_bar:f                        \exp_args:Nf
\module_bar:o                        \exp_args:No
\module_bar:V                        \exp_args:NV
\module_bar:v                        \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

Functional Progarmming (Prg)

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*⟩`}`.

2.2 Collecting Returned Values

`\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`.

Chapter 3

Argument Using (`Use`)

3.1 Expanding 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.

3.2 Using Tokens

`\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 4

Control Structures (`Bool`)

4.1 Constant and Scratch Booleans

`\cTrueBool \cFalseBool`

Constants that represent `true` and `false`, respectively. Used to implement predicates. For example

```
\BoolVarIfTF \cTrueBool {\Result{True!}} {\Result{False!}}
\BoolVarIfTF \cFalseBool {\Result{True!}} {\Result{False!}}
```

True! False!

`\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.

4.2 Creating and Setting 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`.

`\BoolConst` ⟨*boolean*⟩ `{`⟨*boolexpr*⟩`}`

Creates a new constant ⟨*boolean*⟩ or raises an error if the name is already taken. The value of the ⟨*boolean*⟩ is set globally to the result of evaluating the ⟨*boolexpr*⟩. For example

```
\BoolConst \cFooSomeBool {\IntCompare{3}>{2}}
\BoolVarLog \cFooSomeBool
```

`\BoolSet` ⟨*boolean*⟩ {⟨*boolexpr*⟩}

Evaluates the ⟨*boolean expression*⟩ and sets the ⟨*boolean*⟩ variable to the logical truth of this evaluation. For example

```
\BoolSet \lTmpaBool {\IntCompare{3}<{2}}
\BoolVarLog \lTmpaBool
```

`\BoolSetTrue` ⟨*boolean*⟩

Sets ⟨*boolean*⟩ logically `true`.

`\BoolSetFalse` ⟨*boolean*⟩

Sets ⟨*boolean*⟩ logically `false`.

`\BoolSetEq` ⟨*boolean*$_1$⟩ ⟨*boolean*$_2$⟩

Sets ⟨*boolean*$_1$⟩ to the current value of ⟨*boolean*$_2$⟩. For example

```
\BoolSetTrue \lTmpaBool
\BoolSetEq \lTmpbBool \lTmpaBool
\BoolVarLog \lTmpbBool
```

4.3 Viewing Booleans

`\BoolLog` {⟨*boolean expression*⟩}

Writes the logical truth of the ⟨*boolean expression*⟩ in the log file.

`\BoolVarLog` ⟨*boolean*⟩

Writes the logical truth of the ⟨*boolean*⟩ in the log file.

`\BoolShow` {⟨*boolean expression*⟩}

Displays the logical truth of the ⟨*boolean expression*⟩ on the terminal.

`\BoolVarShow` ⟨*boolean*⟩

Displays the logical truth of the ⟨*boolean*⟩ on the terminal.

4.4 Booleans and Conditionals

`\BoolIfExist` ⟨*boolean*⟩
`\BoolIfExistTF` ⟨*boolean*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*boolean*⟩ is currently defined. This does not check that the ⟨*boolean*⟩ really is a boolean variable. For example

```
\BoolIfExistTF \lTmpaBool {\Result{Yes}} {\Result{No}}
\BoolIfExistTF \lFooUndefinedBool {\Result{Yes}} {\Result{No}}
```

Yes No

\BoolVarIf ⟨*boolean*⟩
\BoolVarIfTF ⟨*boolean*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

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

True! False!

\BoolVarNot ⟨*boolean*⟩
\BoolVarNotTF ⟨*boolean*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Evaluates ⟨*true code*⟩ if ⟨*boolean*⟩ is `false`, and ⟨*false code*⟩ If ⟨*boolean*⟩ is `true`. For example

```
\BoolVarNotTF {\IntCompare{3}>{2}} {\Result{Yes}} {\Result{No}}
```

No

\BoolVarAnd ⟨*boolean*$_1$⟩ ⟨*boolean*$_2$⟩
\BoolVarAndTF ⟨*boolean*$_1$⟩ ⟨*boolean*$_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Implements the "And" operation between two booleans, hence is `true` if both are `true`. The ⟨*boolean*$_2$⟩ is only evaluated if it is needed to determine the result of **\BoolVarAnd**. For example

```
\BoolVarAndTF {\IntCompare{3}>{2}} {\IntCompare{3}>{4}} {\Result{Yes}} {\Result{No}}
```

No

\BoolVarOr ⟨*boolean*$_1$⟩ ⟨*boolean*$_2$⟩
\BoolVarOrTF ⟨*boolean*$_1$⟩ ⟨*boolean*$_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Implements the "Or" operation between two booleans, hence is `true` if either one is `true`. The ⟨*boolean*$_2$⟩ is only evaluated if it is needed to determine the result of **\BoolVarOr**. For example

```
\BoolVarOrTF {\IntCompare{3}>{2}} {\IntCompare{3}>{4}} {\Result{Yes}} {\Result{No}}
```

Yes

\BoolVarXor ⟨*boolean*$_1$⟩ ⟨*boolean*$_2$⟩
\BoolVarXorTF ⟨*boolean*$_1$⟩ ⟨*boolean*$_2$⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Implements an "exclusive or" operation between two booleans. For example

```
\BoolVarXorTF {\IntCompare{3}>{2}} {\IntCompare{3}>{4}} {\Result{Yes}} {\Result{No}}
```

Yes

Chapter 5

Token Lists (`Tl`)

5.1 Constant and Scratch Token Lists

`\cSpaceTl`

An explicit space character contained in a token list. For use where an explicit space is required.

`\cEmptyTl`

Constant that is always empty.

`\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.

5.2 Creating and Using 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.

`\TlConst` ⟨*tl var*⟩ `{`⟨*token list*⟩`}`

Creates a new constant ⟨*tl var*⟩ or raises an error if the name is already taken. The value of the ⟨*tl var*⟩ is set globally to the ⟨*token list*⟩.

`\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.

`\TlToStr` {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, returning the resulting character tokens. A ⟨*string*⟩ is a series of tokens with category code 12 (other) with the exception of spaces, which retain category code 10 (space).

`\TlVarToStr` ⟨*tl var*⟩

Converts the content of the ⟨*tl var*⟩ to a string, returning the resulting character tokens. A ⟨*string*⟩ is a series of tokens with category code 12 (other) with the exception of spaces, which retain category code 10 (space).

5.3 Viewing Token Lists

`\TlLog` {⟨*token list*⟩}

Writes the ⟨*token list*⟩ in the log file. See also `\TlShow` which displays the result in the terminal.

`\TlVarLog` ⟨*tl var*⟩

Writes the content of the ⟨*tl var*⟩ in the log file. See also `\TlVarShow` which displays the result in the terminal.

`\TlShow` {⟨*token list*⟩}

Displays the ⟨*token list*⟩ on the terminal.

`\TlVarShow` ⟨*tl var*⟩

Displays the content of the ⟨*tl var*⟩ on the terminal. This is similar to the TeX primitive `\show`, wrapped to a fixed number of characters per line.

5.4 Setting Token List Variables

`\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

`\TlSetEq` ⟨*tl* var_1⟩ ⟨*tl* var_2⟩

Sets the content of ⟨*tl* var_1⟩ equal to that of ⟨*tl* var_2⟩.

`\TlClear` ⟨*tl var*⟩

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

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

Two

\TlClearNew ⟨*tl var*⟩

Ensures that the ⟨*tl var*⟩ exists globally by applying **\TlNew** if necessary, then applies **\TlClear** to leave the ⟨*tl var*⟩ empty.

\TlConcat ⟨*tl var*$_1$⟩ ⟨*tl var*$_2$⟩ ⟨*tl var*$_3$⟩

Concatenates the content of ⟨*tl var*$_2$⟩ and ⟨*tl var*$_3$⟩ together and saves the result in ⟨*tl var*$_1$⟩. The ⟨*tl var*$_2$⟩ is placed at the left side of the new token list.

\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

5.5 Replacing Tokens

Within token lists, replacement takes place at the top level: there is no recursion into brace groups (more precisely, within a group defined by a categroy code 1/2 pair).

\TlReplaceOnce ⟨*tl var*⟩ {⟨*old tokens*⟩} {⟨*new tokens*⟩}

Replaces the first (leftmost) occurrence of ⟨*old tokens*⟩ in the ⟨*tl var*⟩ with ⟨*new tokens*⟩. ⟨*Old tokens*⟩ cannot contain {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

\TlReplaceAll ⟨*tl var*⟩ {⟨*old tokens*⟩} {⟨*new tokens*⟩}

Replaces all occurrences of ⟨*old tokens*⟩ in the ⟨*tl var*⟩ with ⟨*new tokens*⟩. ⟨*Old tokens*⟩ cannot contain {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern ⟨*old tokens*⟩ may remain after the replacement (see **\TlRemoveAll** for an example).

\TlRemoveOnce ⟨*tl var*⟩ {⟨*tokens*⟩}

Removes the first (leftmost) occurrence of ⟨*tokens*⟩ from the ⟨*tl var*⟩. ⟨*Tokens*⟩ cannot contain {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

\TlRemoveAll ⟨*tl var*⟩ {⟨*tokens*⟩}

Removes all occurrences of ⟨*tokens*⟩ from the ⟨*tl var*⟩. ⟨*Tokens*⟩ cannot contain {, } or # (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). As this function operates from left to right, the pattern ⟨*tokens*⟩ may remain after the removal, for instance,

```
\TlSet \lTmpaTl {abbccd}
\TlRemoveAll \lTmpaTl {bc}
\TlUse \lTmpaTl
```

abcd

\TlTrimSpaces {⟨*token list*⟩}

Removes any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from the ⟨*token list*⟩ and returns the result.

\TlVarTrimSpaces ⟨*tl var*⟩

Sets the ⟨*tl var*⟩ to contain the result of removing any leading and trailing explicit space characters (explicit tokens with character code 32 and category code 10) from its contents.

5.6 Working with the Content of Token Lists

\TlCount {⟨*tokens*⟩}

Counts the number of ⟨*items*⟩ in ⟨*tokens*⟩ and returns this information. Unbraced tokens count as one element as do each token group ({···}). This process ignores any unprotected spaces within ⟨*tokens*⟩.

\TlVarCount ⟨*tl var*⟩

Counts the number of ⟨*items*⟩ in the ⟨*tl var*⟩ and returns this information. Unbraced tokens count as one element as do each token group ({···}). This process ignores any unprotected spaces within the ⟨*tl var*⟩.

\TlHead {⟨*token list*⟩}

Returns the first ⟨*item*⟩ in the ⟨*token list*⟩, discarding the rest of the ⟨*token list*⟩. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded; for example

```
\fbox {1\TlHead{ abc }2}
\fbox {1\TlHead{  abc }2}
```

1a2 1a2

If the “head” is a brace group, rather than a single token, the braces are removed, and so

```
\TlHead { { ab} c }
```

yields ␣ab. A blank ⟨*token list*⟩ (see \TlIfBlank) results in \TlHead returning nothing.

\TlVarHead ⟨*tl var*⟩

Returns the first ⟨*item*⟩ in the ⟨*tl var*⟩, discarding the rest of the ⟨*tl var*⟩. All leading explicit space characters (explicit tokens with character code 32 and category code 10) are discarded.

`\TlTail {⟨token list⟩}`

Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first ⟨*item*⟩ in the ⟨*token list*⟩, and returns the remaining tokens. Thus for example

```
\TlTail { a {bc} d }
```

and

```
\TlTail {  a {bc} d }
```

both return ␣{bc}␣d␣. A blank ⟨*token list*⟩ (see `\TlIfBlank`) results in `\TlTail` returning nothing.

`\TlVarTail ⟨tl var⟩`

Discards all leading explicit space characters (explicit tokens with character code 32 and category code 10) and the first ⟨*item*⟩ in the ⟨*tl var*⟩, and returns the remaining tokens.

`\TlItem {⟨token list⟩} {⟨integer expression⟩}`
`\TlVarItem ⟨tl var⟩ {⟨integer expression⟩}`

Indexing items in the ⟨*token list*⟩ from 1 on the left, this function evaluates the ⟨*integer expression*⟩ and returns the appropriate item from the ⟨*token list*⟩. If the ⟨*integer expression*⟩ is negative, indexing occurs from the right of the token list, starting at −1 for the right-most item. If the index is out of bounds, then the function returns nothing.

`\TlRandItem {⟨token list⟩}`
`\TlVarRandItem ⟨tl var⟩`

Selects and returns a pseudo-random item of the ⟨*token list*⟩. If the ⟨*token list*⟩ is blank, the result is empty.

5.7 Mapping over Token Lists

All mappings are done at the current group level, i.e. any local assignments made by the ⟨*function*⟩ or ⟨*code*⟩ discussed below remain in effect after the loop.

`\TlMapInline {⟨token list⟩} {⟨inline function⟩}`

Applies the ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*token list*⟩. The ⟨*inline function*⟩ should consist of code which receives the ⟨*item*⟩ as `#1`.

`\TlVarMapInline ⟨tl var⟩ {⟨inline function⟩}`

Applies the ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*tl var*⟩. The ⟨*inline function*⟩ should consist of code which receives the ⟨*item*⟩ as `#1`.

`\TlMapVariable {⟨token list⟩} ⟨variable⟩ {⟨code⟩}`

Stores each ⟨*item*⟩ of the ⟨*token list*⟩ in turn in the (token list) ⟨*variable*⟩ and applies the ⟨*code*⟩. The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last ⟨*item*⟩ in the ⟨*tl var*⟩, or its original value if the ⟨*tl var*⟩ is blank.

\TlVarMapVariable ⟨*tl var*⟩ ⟨*variable*⟩ {⟨*code*⟩}

Stores each ⟨*item*⟩ of the ⟨*tl var*⟩ in turn in the (token list) ⟨*variable*⟩ and applies the ⟨*code*⟩. The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last ⟨*item*⟩ in the ⟨*tl var*⟩, or its original value if the ⟨*tl var*⟩ is blank.

5.8 Token List Conditionals

\TlIfExist ⟨*tl var*⟩
\TlIfExistTF ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*tl var*⟩ is currently defined. This does not check that the ⟨*tl var*⟩ really is a token list variable.

\TlIfEmpty {⟨*token list*⟩}
\TlIfEmptyTF {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

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

```
\TlIfEmptyTF {abc} {\Result{Empty}} {\Result{NonEmpty}}
\TlIfEmptyTF {} {\Result{Empty}} {\Result{NonEmpty}}
```

NonEmpty Empty

\TlVarIfEmpty ⟨*tl var*⟩
\TlVarIfEmptyTF ⟨*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}
\TlVarIfEmptyTF \lTmpaTl {\Result{Empty}} {\Result{NonEmpty}}
\TlClear \lTmpaTl
\TlVarIfEmptyTF \lTmpaTl {\Result{Empty}} {\Result{NonEmpty}}
```

NonEmpty Empty

\TlIfBlank {⟨*token list*⟩}
\TlIfBlankTF {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list*⟩ consists only of blank spaces (i.e. contains no item). The test is `true` if ⟨*token list*⟩ is zero or more explicit space characters (explicit tokens with character code 32 and category code 10), and is `false` otherwise.

\TlIfEq {⟨*token list*$_1$⟩} {⟨*token list*$_2$⟩}
\TlIfEqTF {⟨*token list*$_1$⟩} {⟨*token list*$_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token list*$_1$⟩ and ⟨*token list*$_2$⟩ contain the same list of tokens, both in respect of character codes and category codes. See **\StrIfEq** if category codes are not important. For example

```
\TlIfEqTF {abc} {abc} {\Result{Yes}} {\Result{No}}
\TlIfEqTF {abc} {xyz} {\Result{Yes}} {\Result{No}}
```

Yes No

\TlVarIfEq ⟨*tl var*$_1$⟩ ⟨*tl var*$_2$⟩
\TlVarIfEqTF ⟨*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}
\TlVarIfEqTF \lTmpaTl \lTmpbTl {\Result{Yes}} {\Result{No}}
\TlVarIfEqTF \lTmpaTl \lTmpcTl {\Result{Yes}} {\Result{No}}
```

Yes No

See also `\StrVarIfEq` for a comparison that ignores category codes.

`\TlIfIn` {⟨*token list*$_1$⟩} {⟨*token list*$_2$⟩}
`\TlIfInTF` {⟨*token list*$_1$⟩} {⟨*token list*$_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if ⟨*token list*$_2$⟩ is found inside ⟨*token list*$_1$⟩. The ⟨*token list*$_2$⟩ cannot contain the tokens `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6). The search does *not* enter brace (category code 1/2) groups.

`\TlVarIfIn` ⟨*tl var*⟩ {⟨*token list*⟩}
`\TlVarIfInTF` ⟨*tl var*⟩ {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list*⟩ is found in the content of the ⟨*tl var*⟩. The ⟨*token list*⟩ cannot contain the tokens `{`, `}` or `#` (more precisely, explicit character tokens with category code 1 (begin-group) or 2 (end-group), and tokens with category code 6).

`\TlIfSingle` {⟨*token list*⟩}
`\TlIfSingleTF` {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*token list*⟩ has exactly one ⟨*item*⟩, i.e. is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to `\TlCount`.

`\TlVarIfSingle` ⟨*tl var*⟩
`\TlVarIfSingleTF` ⟨*tl var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the content of the ⟨*tl var*⟩ consists of a single ⟨*item*⟩, i.e. is a single normal token (neither an explicit space character nor a begin-group character) or a single brace group, surrounded by optional spaces on both sides. In other words, such a token list has token count 1 according to `\TlVarCount`.

5.9 Token List Case Functions

`\TlVarCase` ⟨*test token list variable*⟩
  {
    ⟨*token list variable case*$_1$⟩ {⟨*code case*$_1$⟩}
    ⟨*token list variable case*$_2$⟩ {⟨*code case*$_2$⟩}
    ...
    ⟨*token list variable case*$_n$⟩ {⟨*code case*$_n$⟩}
  }

This function compares the ⟨*test token list variable*⟩ in turn with each of the ⟨*token list variable cases*⟩. If the two are equal (as described for `\TlVarIfEq`) then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. The function does nothing if there is no match.

\TlVarCaseT ⟨*test token list variable*⟩
 {
  ⟨*token list variable case*$_1$⟩ {⟨*code case*$_1$⟩}
  ⟨*token list variable case*$_2$⟩ {⟨*code case*$_2$⟩}
  ...
  ⟨*token list variable case*$_n$⟩ {⟨*code case*$_n$⟩}
 }
 {⟨*true code*⟩}

This function compares the ⟨*test token list variable*⟩ in turn with each of the ⟨*token list variable cases*⟩. If the two are equal (as described for **\TlVarIfEq**) then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case).

\TlVarCaseF ⟨*test token list variable*⟩
 {
  ⟨*token list variable case*$_1$⟩ {⟨*code case*$_1$⟩}
  ⟨*token list variable case*$_2$⟩ {⟨*code case*$_2$⟩}
  ...
  ⟨*token list variable case*$_n$⟩ {⟨*code case*$_n$⟩}
 }
 {⟨*false code*⟩}

This function compares the ⟨*test token list variable*⟩ in turn with each of the ⟨*token list variable cases*⟩. If the two are equal (as described for **\TlVarIfEq**) then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If none match then the ⟨*false code*⟩ is inserted into the input stream (after the code for the appropriate case).

\TlVarCaseTF ⟨*test token list variable*⟩
 {
  ⟨*token list variable case*$_1$⟩ {⟨*code case*$_1$⟩}
  ⟨*token list variable case*$_2$⟩ {⟨*code case*$_2$⟩}
  ...
  ⟨*token list variable case*$_n$⟩ {⟨*code case*$_n$⟩}
 }
 {⟨*true code*⟩}
 {⟨*false code*⟩}

This function compares the ⟨*test token list variable*⟩ in turn with each of the ⟨*token list variable cases*⟩. If the two are equal (as described for **\TlVarIfEq**) then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted. The function **\TlVarCase**, which does nothing if there is no match, is also available.

Chapter 6

Strings (`Str`)

6.1 Constant and Scratch Strings

`\cAmpersandStr \cAtsignStr \cBackslashStr \cLeftBraceStr \cRightBraceStr \cCircumflexStr \cColonStr \cDollarStr \cHashStr \cPercentStr \cTildeStr \cUnderscoreStr \cZeroStr`

Constant strings, containing a single character token, with category code 12.

`\lTmpaStr \lTmpbStr \lTmpcStr \lTmpiStr \lTmpjStr \lTmpkStr`

Scratch strings 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.

`\gTmpaStr \gTmpbStr \gTmpcStr \gTmpiStr \gTmpjStr \gTmpkStr`

Scratch strings 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.

6.2 Creating and Using Strings

`\StrNew` ⟨*str var*⟩

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

`\StrConst` ⟨*str var*⟩ `{`⟨*token list*⟩`}`

Creates a new constant ⟨*str var*⟩ or raises an error if the name is already taken. The value of the ⟨*str var*⟩ is set globally to the ⟨*token list*⟩, converted to a string.

`\StrUse` ⟨*str var*⟩

Recovers the content of a ⟨*str 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 ⟨*str*⟩ directly without an accessor function.

6.3 Viewing Strings

\StrLog {⟨*token list*⟩}

Writes ⟨*token list*⟩ in the log file.

\StrVarLog ⟨*str var*⟩

Writes the content of the ⟨*str var*⟩ in the log file. For example

```
\StrSet \lTmpiStr {1234\abcd5678}
\StrVarLog \lTmpiStr
```

\StrShow {⟨*token list*⟩}

Displays ⟨*token list*⟩ on the terminal.

\StrVarShow ⟨*str var*⟩

Displays the content of the ⟨*str var*⟩ on the terminal.

6.4 Setting String Variables

\StrSet ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and stores the result in ⟨*str var*⟩. For example

```
\StrSet \lTmpiStr {\IntMathMult{4}{5}}
\StrUse \lTmpiStr
```

20

\StrSetEq ⟨*str* var_1⟩ ⟨*str* var_2⟩

Sets the content of ⟨*str* var_1⟩ equal to that of ⟨*str* var_2⟩.

\StrClear ⟨*str var*⟩

Clears the content of the ⟨*str var*⟩. For example

```
\StrSet \lTmpjStr {One}
\StrClear \lTmpjStr
\StrSet \lTmpjStr {Two}
\StrUse \lTmpjStr
```

Two

\StrClearNew ⟨*str var*⟩

Ensures that the ⟨*str var*⟩ exists globally by applying **\StrNew** if necessary, then applies **\StrClear** to leave the ⟨*str var*⟩ empty.

\StrConcat ⟨*str var*$_1$⟩ ⟨*str var*$_2$⟩ ⟨*str var*$_3$⟩

Concatenates the content of ⟨*str var*$_2$⟩ and ⟨*str var*$_3$⟩ together and saves the result in ⟨*str var*$_1$⟩. The ⟨*str var*$_2$⟩ is placed at the left side of the new string variable. The ⟨*str var*$_2$⟩ and ⟨*str var*$_3$⟩ must indeed be strings, as this function does not convert their contents to a string.

\StrPutLeft ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and prepends the result to ⟨*str var*⟩. The current contents of the ⟨*str var*⟩ are not automatically converted to a string. For example

```
\StrSet \lTmpkStr {Functional}
\StrPutLeft \lTmpkStr {Hello}
\StrUse \lTmpkStr
```

HelloFunctional

\StrPutRight ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and appends the result to ⟨*str var*⟩. The current contents of the ⟨*str var*⟩ are not automatically converted to a string. For example

```
\StrSet \lTmpkStr {Functional}
\StrPutRight \lTmpkStr {World}
\StrUse \lTmpkStr
```

FunctionalWorld

6.5 Modifying String Variables

\StrReplaceOnce ⟨*str var*⟩ {⟨*old*⟩} {⟨*new*⟩}

Converts the ⟨*old*⟩ and ⟨*new*⟩ token lists to strings, then replaces the first (leftmost) occurrence of ⟨*old string*⟩ in the ⟨*str var*⟩ with ⟨*new string*⟩.

\StrReplaceAll ⟨*str var*⟩ {⟨*old*⟩} {⟨*new*⟩}

Converts the ⟨*old*⟩ and ⟨*new*⟩ token lists to strings, then replaces all occurrences of ⟨*old string*⟩ in the ⟨*str var*⟩ with ⟨*new string*⟩. As this function operates from left to right, the pattern ⟨*old string*⟩ may remain after the replacement.

\StrRemoveOnce ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ then removes the first (leftmost) occurrence of ⟨*string*⟩ from the ⟨*str var*⟩.

\StrRemoveAll ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ then removes all occurrences of ⟨*string*⟩ from the ⟨*str var*⟩. As this function operates from left to right, the pattern ⟨*string*⟩ may remain after the removal, for instance,

```
\StrSet \lTmpaStr {abbccd}
\StrRemoveAll \lTmpaStr {bc}
\TlUse \lTmpaStr
```

abcd

6.6 Working with the Content of Strings

`\StrCount` {⟨*token list*⟩}

Returns the number of characters in the string representation of ⟨*token list*⟩, as an integer denotation. All characters including spaces are counted.

Due to naming conflict, you need to use `\StrSize` instead of `\StrCount` if you want to use `functional` package together with `xstring` package.

`\StrVarCount` ⟨*tl var*⟩

Returns the number of characters in the string representation of the ⟨*tl var*⟩, as an integer denotation. All characters including spaces are counted.

`\StrHead` {⟨*token list*⟩}

Converts the ⟨*token list*⟩ into a ⟨*string*⟩. The first character in the ⟨*string*⟩ is then returned, with category code "other". If the first character is a space, it returns a space token with category code 10 (blank space). If the ⟨*string*⟩ is empty, then nothing is returned.

`\StrVarHead` ⟨*tl var*⟩

Converts the ⟨*tl var*⟩ into a ⟨*string*⟩. The first character in the ⟨*string*⟩ is then returned, with category code "other". If the first character is a space, it returns a space token with category code 10 (blank space). If the ⟨*string*⟩ is empty, then nothing is returned.

`\StrTail` {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, removes the first character, and returns the remaining characters (if any) with category codes 12 and 10 (for spaces). If the first character is a space, it only trims that space. If the ⟨*token list*⟩ is empty, then nothing is left on the input stream.

`\StrVarTail` ⟨*tl var*⟩

Converts the ⟨*tl var*⟩ to a ⟨*string*⟩, removes the first character, and returns the remaining characters (if any) with category codes 12 and 10 (for spaces). If the first character is a space, it only trims that space. If the ⟨*token list*⟩ is empty, then nothing is left on the input stream.

`\StrItem` {⟨*token list*⟩} {⟨*integer expression*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and returns the character in position ⟨*integer expression*⟩ of the ⟨*string*⟩, starting at 1 for the first (left-most) character. All characters including spaces are taken into account. If the ⟨*integer expression*⟩ is negative, characters are counted from the end of the ⟨*string*⟩. Hence, -1 is the right-most character, *etc.*

`\StrVarItem` ⟨*tl var*⟩ {⟨*integer expression*⟩}

Converts the ⟨*tl var*⟩ to a ⟨*string*⟩, and returns the character in position ⟨*integer expression*⟩ of the ⟨*string*⟩, starting at 1 for the first (left-most) character. All characters including spaces are taken into account. If the ⟨*integer expression*⟩ is negative, characters are counted from the end of the ⟨*string*⟩. Hence, -1 is the right-most character, *etc.*

6.7 Mapping over Strings

All mappings are done at the current group level, *i.e.* any local assignments made by the ⟨*function*⟩ or ⟨*code*⟩ discussed below remain in effect after the loop.

\StrMapInline {⟨*token list*⟩} {⟨*inline function*⟩}
\StrVarMapInline ⟨*str var*⟩ {⟨*inline function*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ then applies the ⟨*inline function*⟩ to every ⟨*character*⟩ in the ⟨*str var*⟩ including spaces. The ⟨*inline function*⟩ should consist of code which receives the ⟨*character*⟩ as `#1`.

\StrMapVariable {⟨*token list*⟩} ⟨*variable*⟩ {⟨*code*⟩}
\StrVarMapVariable ⟨*str var*⟩ ⟨*variable*⟩ {⟨*code*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ then stores each ⟨*character*⟩ in the ⟨*string*⟩ (including spaces) in turn in the (string or token list) ⟨*variable*⟩ and applies the ⟨*code*⟩. The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last ⟨*character*⟩ in the ⟨*string*⟩, or its original value if the ⟨*string*⟩ is empty.

6.8 String Conditionals

\StrIfExist ⟨*str var*⟩
\StrIfExistTF ⟨*str var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*str var*⟩ is currently defined. This does not check that the ⟨*str var*⟩ really is a string.

\StrVarIfEmpty ⟨*str var*⟩
\StrVarIfEmptyTF ⟨*str var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

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

```
\StrSet \lTmpaStr {abc}
\StrVarIfEmptyTF \lTmpaStr {\Result{Empty}} {\Result{NonEmpty}}
\StrClear \lTmpaStr
\StrVarIfEmptyTF \lTmpaStr {\Result{Empty}} {\Result{NonEmpty}}
```

NonEmpty Empty

\StrIfEq {⟨tl_1⟩} {⟨tl_2⟩}
\StrIfEqTF {⟨tl_1⟩} {⟨tl_2⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Compares the two ⟨*token lists*⟩ on a character by character basis (namely after converting them to strings), and is `true` if the two ⟨*strings*⟩ contain the same characters in the same order. See **\TlIfEq** to compare tokens (including their category codes) rather than characters. For example

```
\StrIfEqTF {abc} {abc} {\Result{Yes}} {\Result{No}}
\StrIfEqTF {abc} {xyz} {\Result{Yes}} {\Result{No}}
```

Yes No

\StrVarIfEq ⟨*str* var_1⟩ ⟨*str* var_2⟩
\StrVarIfEqTF ⟨*str* var_1⟩ ⟨*str* var_2⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Compares the content of two ⟨*str variables*⟩ and is logically `true` if the two contain the same characters in the same order. See **\TlVarIfEq** to compare tokens (including their category codes) rather than characters.

```
\StrSet \lTmpaStr {abc}
\StrSet \lTmpbStr {abc}
\StrSet \lTmpcStr {xyz}
\StrVarIfEqTF \lTmpaStr \lTmpbStr {\Result{Yes}} {\Result{No}}
\StrVarIfEqTF \lTmpaStr \lTmpcStr {\Result{Yes}} {\Result{No}}
```

Yes No

\StrIfInTF {⟨tl_1⟩} {⟨tl_2⟩}
\StrIfInTF {⟨tl_1⟩} {⟨tl_2⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Converts both ⟨*token lists*⟩ to ⟨*strings*⟩ and tests whether ⟨$string_2$⟩ is found inside ⟨$string_1$⟩.

\StrVarIfInTF ⟨*str var*⟩ {⟨*token list*⟩}
\StrVarIfInTF ⟨*str var*⟩ {⟨*token list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ and tests if that ⟨*string*⟩ is found in the content of the ⟨*str var*⟩.

\StrCompare {⟨tl_1⟩} ⟨*relation*⟩ {⟨tl_2⟩}
\StrCompareTF {⟨tl_1⟩} ⟨*relation*⟩ {⟨tl_2⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Compares the two ⟨*token lists*⟩ on a character by character basis (namely after converting them to strings) in a lexicographic order according to the character codes of the characters. The ⟨*relation*⟩ can be `<`, `=`, or `>` and the test is `true` under the following conditions:

- for `<`, if the first string is earlier than the second in lexicographic order;
- for `=`, if the two strings have exactly the same characters;
- for `>`, if the first string is later than the second in lexicographic order.

For example:

```
\StrCompareTF {ab} < {abc} {\Result{Yes}} {\Result{No}}
\StrCompareTF {ab} < {aa} {\Result{Yes}} {\Result{No}}
```

Yes No

Due to naming conflict, you need to use **\StrIfCompare**/**\StrIfCompareTF** as a replacement if you want to use `functional` package together with `xstring` package.

6.9 String Case Functions

```
\StrCase {⟨test string⟩}
  {
    {⟨string case1⟩} {⟨code case1⟩}
    {⟨string case2⟩} {⟨code case2⟩}
    ...
    {⟨string casen⟩} {⟨code casen⟩}
  }
```

Compares the ⟨*test string*⟩ in turn with each of the ⟨*string cases*⟩ (all token lists are converted to strings). If the two are equal (as described for **\StrIfEq**) then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded.

`\StrCaseT` {⟨*test string*⟩}
  {
    {⟨*string case*$_1$⟩} {⟨*code case*$_1$⟩}
    {⟨*string case*$_2$⟩} {⟨*code case*$_2$⟩}
    ...
    {⟨*string case*$_n$⟩} {⟨*code case*$_n$⟩}
  }
  {⟨*true code*⟩}

Compares the ⟨*test string*⟩ in turn with each of the ⟨*string cases*⟩ (all token lists are converted to strings). If the two are equal (as described for `\StrIfEq`) then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case).

`\StrCaseF` {⟨*test string*⟩}
  {
    {⟨*string case*$_1$⟩} {⟨*code case*$_1$⟩}
    {⟨*string case*$_2$⟩} {⟨*code case*$_2$⟩}
    ...
    {⟨*string case*$_n$⟩} {⟨*code case*$_n$⟩}
  }
  {⟨*false code*⟩}

Compares the ⟨*test string*⟩ in turn with each of the ⟨*string cases*⟩ (all token lists are converted to strings). If the two are equal (as described for `\StrIfEq`) then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If none match then the ⟨*false code*⟩ is inserted.

`\StrCaseTF` {⟨*test string*⟩}
  {
    {⟨*string case*$_1$⟩} {⟨*code case*$_1$⟩}
    {⟨*string case*$_2$⟩} {⟨*code case*$_2$⟩}
    ...
    {⟨*string case*$_n$⟩} {⟨*code case*$_n$⟩}
  }
  {⟨*true code*⟩}
  {⟨*false code*⟩}

Compares the ⟨*test string*⟩ in turn with each of the ⟨*string cases*⟩ (all token lists are converted to strings). If the two are equal (as described for `\StrIfEq`) then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted.

Chapter 7

Integers (Int)

7.1 Constant and Scratch Integers

`\cZeroInt` `\cOneInt`

Integer values used with primitive tests and assignments: their self-terminating nature makes these more convenient and faster than literal numbers.

`\cMaxInt`

The maximum value that can be stored as an integer.

`\cMaxRegisterInt`

Maximum number of registers.

`\cMaxCharInt`

Maximum character code completely supported by the engine.

`\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.

7.2 Integer Expressions

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

Evaluates the two ⟨*integer expressions*⟩ as described earlier, then divides the first value by the second, and rounds the result to the closest integer. Ties are rounded away from zero. Note that this is identical to using `/` directly in an ⟨*integer expression*⟩. The result is returned as an ⟨*integer denotation*⟩. For example

```
\IntMathDiv {8} {3}
```
3

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

Evaluates the two ⟨*integer expressions*⟩ as described earlier, then divides the first value by the second, and rounds the result towards zero. Note that division using `/` rounds to the closest integer instead. The result is returned as an ⟨*integer denotation*⟩. For example

```
\IntMathDivTruncate {8} {3}
```
2

`\IntMathSign` {⟨*intexpr*⟩}

Evaluates the ⟨*integer expression*⟩ then leaves 1 or 0 or -1 in the input stream according to the sign of the result.

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

Evaluates the ⟨*integer expression*⟩ as described for `\IntEval` and leaves the absolute value of the result in the input stream as an ⟨*integer denotation*⟩ after two expansions.

`\IntMathMax` {⟨*intexpr*$_1$⟩} {⟨*intexpr*$_2$⟩}
`\IntMathMin` {⟨*intexpr*$_1$⟩} {⟨*intexpr*$_2$⟩}

Evaluates the ⟨*integer expressions*⟩ as described for `\IntEval` and leaves either the larger or smaller value in the input stream as an ⟨*integer denotation*⟩ after two expansions.

\IntMathMod {⟨*intexpr*$_1$⟩} {⟨*intexpr*$_2$⟩}

Evaluates the two ⟨*integer expressions*⟩ as described earlier, then calculates the integer remainder of dividing the first expression by the second. This is obtained by subtracting \IntMathDivTruncate {⟨*intexpr*$_1$⟩} {⟨*intexpr*$_2$⟩} times ⟨*intexpr*$_2$⟩ from ⟨*intexpr*$_1$⟩. Thus, the result has the same sign as ⟨*intexpr*$_1$⟩ and its absolute value is strictly less than that of ⟨*intexpr*$_2$⟩. The result is left in the input stream as an ⟨*integer denotation*⟩ after two expansions.

\IntMathRand {⟨*intexpr*$_1$⟩} {⟨*intexpr*$_2$⟩}

Evaluates the two ⟨*integer expressions*⟩ and produces a pseudo-random number between the two (with bounds included).

7.3 Creating and Using Integers

\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.

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

Creates a new constant ⟨*integer*⟩ or raises an error if the name is already taken. The value of the ⟨*integer*⟩ is set globally to the ⟨*integer expression*⟩.

\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.

7.4 Viewing Integers

\IntLog {⟨*integer expression*⟩}

Writes the result of evaluating the ⟨*integer expression*⟩ in the log file.

\IntVarLog ⟨*integer*⟩

Writes the value of the ⟨*integer*⟩ in the log file.

\IntShow {⟨*integer expression*⟩}

Displays the result of evaluating the ⟨*integer expression*⟩ on the terminal.

\IntVarShow ⟨*integer*⟩

Displays the value of the ⟨*integer*⟩ on the terminal.

7.5 Setting Integer Variables

\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

\IntSetEq ⟨$integer_1$⟩ ⟨$integer_2$⟩

Sets the content of ⟨$integer_1$⟩ equal to that of ⟨$integer_2$⟩.

\IntZero ⟨*integer*⟩

Sets ⟨*integer*⟩ to 0. For example

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

0

\IntZeroNew ⟨*integer*⟩

Ensures that the ⟨*integer*⟩ exists globally by applying **\IntNew** if necessary, then applies **\IntZero** to leave the ⟨*integer*⟩ set to zero.

\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

7.6 Integer Step Functions

\IntStepInline {⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} {⟨*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 inserted into the input stream with `#1` replaced by the current ⟨*value*⟩. Thus the ⟨*code*⟩ should define a function of one argument (`#1`). For example

```
\IgnoreSpacesOn
\TlClear \lTmpaTl
\IntStepInline {1} {3} {30} {
  \TlPutRight \lTmpaTl {[#1]}
}
\Result {\Value\lTmpaTl}
\IgnoreSpacesOff
```

produces [1][4][7][10][13][16][19][22][25][28].

\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*⟩.

7.7 Integer Conditionals

\IntIfExist ⟨*integer*⟩
\IntIfExistTF ⟨*integer*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*integer*⟩ is currently defined. This does not check that the ⟨*integer*⟩ really is an integer variable.

\IntIfOdd {⟨*integer expression*⟩}
\IntIfOddTF {⟨*integer expression*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

This function first evaluates the ⟨*integer expression*⟩ as described for \IntEval. It then evaluates if this is odd or even, as appropriate.

\IntIfEven {⟨*integer expression*⟩}
\IntIfEvenTF {⟨*integer expression*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

This function first evaluates the ⟨*integer expression*⟩ as described for \IntEval. It then evaluates if this is even or odd, as appropriate.

`\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

7.8 Integer Case Functions

```
\IntCase {⟨test integer expression⟩}
  {
    {⟨intexpr case1⟩} {⟨code case1⟩}
    {⟨intexpr case2⟩} {⟨code case2⟩}
    ...
    {⟨intexpr casen⟩} {⟨code casen⟩}
  }
```

This function evaluates the ⟨*test integer expression*⟩ and compares this in turn to each of the ⟨*integer expression cases*⟩. If the two are equal then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded.

```
\IntCaseT {⟨test integer expression⟩}
  {
    {⟨intexpr case1⟩} {⟨code case1⟩}
    {⟨intexpr case2⟩} {⟨code case2⟩}
    ...
    {⟨intexpr casen⟩} {⟨code casen⟩}
  }
  {⟨true code⟩}
```

This function evaluates the ⟨*test integer expression*⟩ and compares this in turn to each of the ⟨*integer expression cases*⟩. If the two are equal then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case).

```
\IntCaseF {⟨test integer expression⟩}
  {
    {⟨intexpr case1⟩} {⟨code case1⟩}
    {⟨intexpr case2⟩} {⟨code case2⟩}
    ...
    {⟨intexpr casen⟩} {⟨code casen⟩}
  }
  {⟨false code⟩}
```

This function evaluates the ⟨*test integer expression*⟩ and compares this in turn to each of the ⟨*integer expression cases*⟩. If the two are equal then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If none match then the ⟨*false code*⟩ is into the input stream (after the code for the appropriate case). For example

```
\IgnoreSpacesOn
\IntCaseF { 2 * 5 }
  {
    { 5 }       { Small }
    { 4 + 6 }   { Medium }
    { -2 * 10 } { Negative }
  }
  { No idea! }
\IgnoreSpacesOff
```

Medium

\IntCaseTF {⟨*test integer expression*⟩}
  {
    {⟨*intexpr case*$_1$⟩} {⟨*code case*$_1$⟩}
    {⟨*intexpr case*$_2$⟩} {⟨*code case*$_2$⟩}
    ...
    {⟨*intexpr case*$_n$⟩} {⟨*code case*$_n$⟩}
  }
  {⟨*true code*⟩}
  {⟨*false code*⟩}

This function evaluates the ⟨*test integer expression*⟩ and compares this in turn to each of the ⟨*integer expression cases*⟩. If the two are equal then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted.

Chapter 8

Floating Point Numbers (`Fp`)

8.1 Constant and Scratch Floating Points

`\cZeroFp \cMinusZeroFp`

Zero, with either sign.

`\cOneFp`

One as an `fp`: useful for comparisons in some places.

`\cInfFp \cMinusInfFp`

Infinity, with either sign. These can be input directly in a floating point expression as `inf` and `-inf`.

`\cEFp`

The value of the base of the natural logarithm, $\mathrm{e} = \exp(1)$.

`\cPiFp`

The value of π. This can be input directly in a floating point expression as `pi`.

`\cOneDegreeFp`

The value of 1° in radians. Multiply an angle given in degrees by this value to obtain a result in radians. Note that trigonometric functions expecting an argument in radians or in degrees are both available. Within floating point expressions, this can be accessed as `deg`.

`\lTmpaFp \lTmpbFp \lTmpcFp \lTmpiFp \lTmpjFp \lTmpkFp`

Scratch floating point numbers 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.

`\gTmpaFp \gTmpbFp \gTmpcFp \gTmpiFp \gTmpjFp \gTmpkFp`

Scratch floating point numbers 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.

8.2 Floating Point Expressions

\FpEval {⟨*floating point expression*⟩}

Evaluates the ⟨*floating point expression*⟩ and returns the result as a decimal number with no exponent. Leading or trailing zeros may be inserted to compensate for the exponent. Non-significant trailing zeros are trimmed, and integers are expressed without a decimal separator. The values $\pm\infty$ and `NaN` trigger an "invalid operation" exception. For a tuple, each item is converted using **\FpEval** and they are combined as `(`⟨fp_1⟩`,␣`⟨fp_2⟩`,␣`…⟨fp_n⟩`)` if $n > 1$ and `(`⟨fp_1⟩`,)` or `()` for fewer items. For example

```
\FpEval {(1.2+3.4)*(5.6-7.8)/9}
```

-1.1244444444444444

\FpMathAdd {⟨$fpexpr_1$⟩} {⟨$fpexpr_2$⟩}

Adds {⟨$fpexpr_1$⟩} and {⟨$fpexpr_2$⟩}, and returns the result. For example

```
\FpMathAdd {2.8} {3.7}
\FpMathAdd {3.8-1} {2.7+1}
```

6.5 6.5

\FpMathSub {⟨$fpexpr_1$⟩} {⟨$fpexpr_2$⟩}

Subtracts {⟨$fpexpr_2$⟩} from {⟨$fpexpr_1$⟩}, and returns the result. For example

```
\FpMathSub {2.8} {3.7}
\FpMathSub {3.8-1} {2.7+1}
```

-0.9 -0.9

\FpMathMult {⟨$fpexpr_1$⟩} {⟨$fpexpr_2$⟩}

Multiplies {⟨$fpexpr_1$⟩} by {⟨$fpexpr_2$⟩}, and returns the result. For example

```
\FpMathMult {2.8} {3.7}
\FpMathMult {3.8-1} {2.7+1}
```

10.36 10.36

\FpMathDiv {⟨$fpexpr_1$⟩} {⟨$fpexpr_2$⟩}

Divides {⟨$fpexpr_1$⟩} by {⟨$fpexpr_2$⟩}, and returns the result. For example

```
\FpMathDiv {2.8} {3.7}
\FpMathDiv {3.8-1} {2.7+1}
```

0.7567567567567568 0.7567567567567568

\FpMathSign {⟨*fpexpr*⟩}

Evaluates the ⟨*fpexpr*⟩ and returns the value using **\FpEval**`{sign(`⟨*result*⟩`)}`: $+1$ for positive numbers and for $+\infty$, -1 for negative numbers and for $-\infty$, ± 0 for ± 0. If the operand is a tuple or is `NaN`, then "invalid operation" occurs and the result is 0. For example

```
\FpMathSign {3.5}
\FpMathSign {-2.7}
```

1 -1

\FpMathAbs {⟨*floating point expression*⟩}

Evaluates the ⟨*floating point expression*⟩ as described for **\FpEval** and returns the absolute value. If the argument is $\pm\infty$, `NaN` or a tuple, "invalid operation" occurs. Within floating point expressions, `abs()` can be used; it accepts $\pm\infty$ and `NaN` as arguments.

\FpMathMax {⟨*fp expression*$_1$⟩} {⟨*fp expression*$_2$⟩}
\FpMathMin {⟨*fp expression*$_1$⟩} {⟨*fp expression*$_2$⟩}

Evaluates the ⟨*floating point expressions*⟩ as described for **\FpEval** and returns the resulting larger (`max`) or smaller (`min`) value. If the argument is a tuple, "invalid operation" occurs, but no other case raises exceptions. Within floating point expressions, `max()` and `min()` can be used.

8.3 Creating and Using Floating Points

\FpNew ⟨*fp var*⟩

Creates a new ⟨*fp var*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*fp var*⟩ is initially +0.

\FpConst ⟨*fp var*⟩ {⟨*floating point expression*⟩}

Creates a new constant ⟨*fp var*⟩ or raises an error if the name is already taken. The ⟨*fp var*⟩ is set globally equal to the result of evaluating the ⟨*floating point expression*⟩. For example

```
\FpConst \cMyPiFp {3.1415926}
\FpUse \cMyPiFp
```

3.1415926

\FpUse ⟨*fp var*⟩

Recovers the value of the ⟨*fp var*⟩ and returns the value as a decimal number with no exponent.

8.4 Viewing Floating Points

\FpLog {⟨*floating point expression*⟩}

Evaluates the ⟨*floating point expression*⟩ and writes the result in the log file.

\FpVarLog ⟨*fp var*⟩

Writes the value of ⟨*fp var*⟩ in the log file.

\FpShow {⟨*floating point expression*⟩}

Evaluates the ⟨*floating point expression*⟩ and displays the result in the terminal.

\FpVarShow ⟨*fp var*⟩

Displays the value of ⟨*fp var*⟩ in the terminal.

8.5 Setting Floating Point Variables

`\FpSet` ⟨*fp var*⟩ {⟨*floating point expression*⟩}

Sets ⟨*fp var*⟩ equal to the result of computing the ⟨*floating point expression*⟩. For example

```
\FpSet \lTmpaFp {4/7}
\FpUse \lTmpaFp
```

0.5714285714285714

`\FpSetEq` ⟨*fp* var_1⟩ ⟨*fp* var_2⟩

Sets the floating point variable ⟨*fp* var_1⟩ equal to the current value of ⟨*fp* var_2⟩.

`\FpZero` ⟨*fp var*⟩

Sets the ⟨*fp var*⟩ to +0. For example

```
\FpSet \lTmpaFp {5.3}
\FpZero \lTmpaFp
\FpUse \lTmpaFp
```

0

`\FpZeroNew` ⟨*fp var*⟩

Ensures that the ⟨*fp var*⟩ exists globally by applying `\FpNew` if necessary, then applies `\FpZero` to leave the ⟨*fp var*⟩ set to +0.

`\FpAdd` ⟨*fp var*⟩ {⟨*floating point expression*⟩}

Adds the result of computing the ⟨*floating point expression*⟩ to the ⟨*fp var*⟩. This also applies if ⟨*fp var*⟩ and ⟨*floating point expression*⟩ evaluate to tuples of the same size. For example

```
\FpSet \lTmpaFp {5.3}
\FpAdd \lTmpaFp {2.11}
\FpUse \lTmpaFp
```

7.41

`\FpSub` ⟨*fp var*⟩ {⟨*floating point expression*⟩}

Subtracts the result of computing the ⟨*floating point expression*⟩ from the ⟨*fp var*⟩. This also applies if ⟨*fp var*⟩ and ⟨*floating point expression*⟩ evaluate to tuples of the same size. For example

```
\FpSet \lTmpaFp {5.3}
\FpSub \lTmpaFp {2.11}
\FpUse \lTmpaFp
```

3.19

8.6 Floating Point Step Functions

`\FpStepInline` {⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} {⟨*code*⟩}

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each ⟨*value*⟩ from the

⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream with `#1` replaced by the current ⟨*value*⟩. Thus the ⟨*code*⟩ should define a function of one argument (`#1`). For example

```
\IgnoreSpacesOn
\TlClear \lTmpaTl
\FpStepInline {1} {0.1} {1.5} {
  \TlPutRight \lTmpaTl {[#1]}
}
\Result {\Value\lTmpaTl}
\IgnoreSpacesOff
```

produces [1][1.1][1.2][1.3][1.4][1.5].

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

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be floating point expressions evaluating to a floating point number, not a tuple. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream, with the ⟨*tl var*⟩ defined as the current ⟨*value*⟩. Thus the ⟨*code*⟩ should make use of the ⟨*tl var*⟩.

8.7 Float Point Conditionals

`\FpIfExist` ⟨*fp var*⟩
`\FpIfExistTF` ⟨*fp var*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests whether the ⟨*fp var*⟩ is currently defined. This does not check that the ⟨*fp var*⟩ really is a floating point variable. For example

```
\FpIfExistTF \lTmpaFp {\Result{Yes}} {\Result{No}}
\FpIfExistTF \lMyUndefinedFp {\Result{Yes}} {\Result{No}}
```

Yes No

`\FpCompare` `{`⟨$fpexpr_1$⟩`}` ⟨*relation*⟩ `{`⟨$fpexpr_2$⟩`}`
`\FpCompareTF` `{`⟨$fpexpr_1$⟩`}` ⟨*relation*⟩ `{`⟨$fpexpr_2$⟩`}` `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Compares the ⟨$fpexpr_1$⟩ and the ⟨$fpexpr_2$⟩, and returns `true` if the ⟨*relation*⟩ is obeyed. For example

```
\FpCompareTF {1} > {0.9999} {\Result{Greater}} {\Result{Less}}
\FpCompareTF {1} > {1.0001} {\Result{Greater}} {\Result{Less}}
```

Greater Less

Two floating points x and y may obey four mutually exclusive relations: $x < y$, $x = y$, $x > y$, or $x?y$ (“not ordered”). The last case occurs exactly if one or both operands is `NaN` or is a tuple, unless they are equal tuples. Note that a `NaN` is distinct from any value, even another `NaN`, hence $x = x$ is not true for a `NaN`. To test if a value is `NaN`, compare it to an arbitrary number with the “not ordered” relation.

Tuples are equal if they have the same number of items and items compare equal (in particular there must be no `NaN`). At present any other comparison with tuples yields `?` (not ordered). This is experimental.

Chapter 9

Dimensions (Dim)

9.1 Constant and Scratch Dimensions

`\cMaxDim`

The maximum value that can be stored as a dimension. This can also be used as a component of a skip.

`\cZeroDim`

A zero length as a dimension. This can also be used as a component of a skip.

`\lTmpaDim \lTmpbDim \lTmpcDim \lTmpiDim \lTmpjDim \lTmpkDim`

Scratch dimensions 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.

`\gTmpaDim \gTmpbDim \gTmpcDim \gTmpiDim \gTmpjDim \gTmpkDim`

Scratch dimensions 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.

9.2 Dimension Expressions

`\DimEval` {⟨*dimension expression*⟩}

Evaluates the ⟨*dimension expression*⟩, expanding any dimensions and token list variables within the ⟨*expression*⟩ to their content (without requiring `\DimUse`/`\TlUse`) and applying the standard mathematical rules. The result of the calculation is returned as a ⟨*dimension denotation*⟩. For example

```
\DimEval {(1.2pt+3.4pt)/9}
```

0.51111pt

`\DimMathAdd` {⟨$dimexpr_1$⟩} {⟨$dimexpr_2$⟩}

Adds {⟨$dimexpr_1$⟩} and {⟨$dimexpr_2$⟩}, and returns the result. For example

```
\DimMathAdd {2.8pt} {3.7pt}
\DimMathAdd {3.8pt-1pt} {2.7pt+1pt}
```

6.5pt 6.5pt

\DimMathSub {⟨*dimexpr*$_1$⟩} {⟨*dimexpr*$_2$⟩}

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

```
\DimMathSub {2.8pt} {3.7pt}
\DimMathSub {3.8pt-1pt} {2.7pt+1pt}
```

-0.9pt -0.9pt

\DimMathRatio {⟨*dimexpr*$_1$⟩} {⟨*dimexpr*$_2$⟩}

Parses the two ⟨*dimension expressions*⟩, then calculates the ratio of the two and returns it. The result is a ratio expression between two integers, with all distances converted to scaled points. For example

```
\DimMathRatio {5pt} {10pt}
```

327680/655360

The returned value is suitable for use inside a ⟨*dimension expression*⟩ such as

```
\DimSet \lTmpaDim {10pt*\DimMathRatio{5pt}{10pt}}
```

\DimMathSign {⟨*dimexpr*⟩}

Evaluates the ⟨*dimexpr*⟩ then returns 1 or 0 or −1 according to the sign of the result. For example

```
\DimMathSign {3.5pt}
\DimMathSign {-2.7pt}
```

1 -1

\DimMathAbs {⟨*dimexpr*⟩}

Converts the ⟨*dimexpr*⟩ to its absolute value, returning the result as a ⟨*dimension denotation*⟩. For example

```
\DimMathAbs {3.5pt}
\DimMathAbs {-2.7pt}
```

3.5pt 2.7pt

\DimMathMax {⟨*dimexpr*$_1$⟩} {⟨*dimexpr*$_2$⟩}
\DimMathMin {⟨*dimexpr*$_1$⟩} {⟨*dimexpr*$_2$⟩}

Evaluates the two ⟨*dimension expressions*⟩ and returns either the maximum or minimum value as appropriate as a ⟨*dimension denotation*⟩. For example

```
\DimMathMax {3.5pt} {-2.7pt}
\DimMathMin {3.5pt} {-2.7pt}
```

3.5pt -2.7pt

9.3 Creating and Using Dimensions

\DimNew ⟨*dimension*⟩

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

\DimConst ⟨*dimension*⟩ {⟨*dimension expression*⟩}

Creates a new constant ⟨*dimension*⟩ or raises an error if the name is already taken. The value of the ⟨*dimension*⟩ is set globally to the ⟨*dimension expression*⟩. For example

```
\DimConst \cFooSomeDim {1cm}
\DimUse \cFooSomeDim
```

28.45274pt

\DimUse ⟨*dimension*⟩

Recovers the content of a ⟨*dimension*⟩ and returns the value. An error is raised if the variable does not exist or if it is invalid.

9.4 Viewing Dimensions

\DimLog {⟨*dimension expression*⟩}

Writes the result of evaluating the ⟨*dimension expression*⟩ in the log file. For example

```
\DimLog {\lFooSomeDim+1cm}
```

\DimVarLog ⟨*dimension*⟩

Writes the value of the ⟨*dimension*⟩ in the log file. For example

```
\DimVarLog \lFooSomeDim
```

\DimShow {⟨*dimension expression*⟩}

Displays the result of evaluating the ⟨*dimension expression*⟩ on the terminal. For example

```
\DimShow {\lFooSomeDim+1cm}
```

\DimVarShow ⟨*dimension*⟩

Displays the value of the ⟨*dimension*⟩ on the terminal. For example

```
\DimVarShow \lFooSomeDim
```

9.5 Setting Dimension Variables

\DimSet ⟨*dimension*⟩ {⟨*dimension expression*⟩}

Sets ⟨*dimension*⟩ to the value of ⟨*dimension expression*⟩, which must evaluate to a length with units.

\DimSetEq ⟨$dimension_1$⟩ ⟨$dimension_2$⟩

Sets the content of ⟨$dimension_1$⟩ equal to that of ⟨$dimension_2$⟩. For example

```
\DimSet \lTmpaDim {10pt}
\DimSetEq \lTmpbDim \lTmpaDim
\DimUse \lTmpbDim
```

10.0pt

`\DimZero` ⟨*dimension*⟩

Sets ⟨*dimension*⟩ to 0 pt. For example

```
\DimSet \lTmpaDim {1em}
\DimZero \lTmpaDim
\DimUse \lTmpaDim
```

0.0pt

`\DimZeroNew` ⟨*dimension*⟩

Ensures that the ⟨*dimension*⟩ exists globally by applying `\DimNew` if necessary, then applies `\DimZero` to set the ⟨*dimension*⟩ to zero. For example

```
\DimZeroNew \lFooSomeDim
\DimUse \lFooSomeDim
```

0.0pt

`\DimAdd` ⟨*dimension*⟩ {⟨*dimension expression*⟩}

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

```
\DimSet \lTmpaDim {5.3pt}
\DimAdd \lTmpaDim {2.11pt}
\DimUse \lTmpaDim
```

7.41pt

`\DimSub` ⟨*dimension*⟩ {⟨*dimension expression*⟩}

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

```
\DimSet \lTmpaDim {5.3pt}
\DimSub \lTmpaDim {2.11pt}
\DimUse \lTmpaDim
```

3.19pt

9.6 Dimension Step Functions

`\DimStepInline` {⟨*initial value*⟩} {⟨*step*⟩} {⟨*final value*⟩} {⟨*code*⟩}

This function first evaluates the ⟨*initial value*⟩, ⟨*step*⟩ and ⟨*final value*⟩, all of which should be dimension expressions. Then for each ⟨*value*⟩ from the ⟨*initial value*⟩ to the ⟨*final value*⟩ in turn (using ⟨*step*⟩ between each ⟨*value*⟩), the ⟨*code*⟩ is inserted into the input stream with `#1` replaced by the current ⟨*value*⟩. Thus the ⟨*code*⟩ should define a function of one argument (`#1`). For example

```
\IgnoreSpacesOn
\TlClear \lTmpaTl
\DimStepInline {1pt} {0.1pt} {1.5pt} {
  \TlPutRight \lTmpaTl {[#1]}
}
\Result {\Value\lTmpaTl}
\IgnoreSpacesOff
```

produces [1.0pt][1.1pt][1.20001pt][1.30002pt][1.40002pt].

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

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

9.7 Dimension Conditionals

\DimIfExist ⟨*dimension*⟩
\DimIfExistTF ⟨*dimension*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*dimension*⟩ is currently defined. This does not check that the ⟨*dimension*⟩ really is a dimension variable. For example

```
\DimIfExistTF \lTmpaDim {\Result{Yes}} {\Result{No}}
\DimIfExistTF \lFooUndefinedDim {\Result{Yes}} {\Result{No}}
```

Yes No

\DimCompare {⟨*dimexpr*$_1$⟩} ⟨*relation*⟩ {⟨*dimexpr*$_2$⟩}
\DimCompareTF {⟨*dimexpr*$_1$⟩} ⟨*relation*⟩ {⟨*dimexpr*$_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}

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

Equal	=
Greater than	>
Less than	<

For example

```
\DimCompareTF {1pt} > {0.9999pt} {\Result{Greater}} {\Result{Less}}
\DimCompareTF {1pt} > {1.0001pt} {\Result{Greater}} {\Result{Less}}
```

Greater Less

9.8 Dimension Case Functions

```
\DimCase {⟨test dimension expression⟩}
  {
    {⟨dimexpr case1⟩} {⟨code case1⟩}
    {⟨dimexpr case2⟩} {⟨code case2⟩}
    ...
    {⟨dimexpr casen⟩} {⟨code casen⟩}
  }
```

This function evaluates the ⟨*test dimension expression*⟩ and compares this in turn to each of the ⟨*dimension expression cases*⟩. If the two are equal then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded.

```
\DimCaseT {⟨test dimension expression⟩}
  {
    {⟨dimexpr case1⟩} {⟨code case1⟩}
    {⟨dimexpr case2⟩} {⟨code case2⟩}
    ...
    {⟨dimexpr casen⟩} {⟨code casen⟩}
  }
  {⟨true code⟩}
```

This function evaluates the ⟨*test dimension expression*⟩ and compares this in turn to each of the ⟨*dimension expression cases*⟩. If the two are equal then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case).

```
\DimCaseF {⟨test dimension expression⟩}
  {
    {⟨dimexpr case1⟩} {⟨code case1⟩}
    {⟨dimexpr case2⟩} {⟨code case2⟩}
    ...
    {⟨dimexpr casen⟩} {⟨code casen⟩}
  }
  {⟨false code⟩}
```

This function evaluates the ⟨*test dimension expression*⟩ and compares this in turn to each of the ⟨*dimension expression cases*⟩. If the two are equal then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If none of the cases match then the ⟨*false code*⟩ is inserted. For example

```
\IgnoreSpacesOn
\DimSet \lTmpaDim {5pt}
\DimCaseF {2\lTmpaDim} {
  {5pt}     {\Result{Small}}
  {4pt+6pt} {\Result{Medium}}
  {-10pt}   {\Result{Negative}}
}{
  \Result {No Match}
}
\IgnoreSpacesOff
```

Medium

```
\DimCaseTF {⟨test dimension expression⟩}
  {
    {⟨dimexpr case1⟩} {⟨code case1⟩}
    {⟨dimexpr case2⟩} {⟨code case2⟩}
    …
    {⟨dimexpr casen⟩} {⟨code casen⟩}
  }
  {⟨true code⟩}
  {⟨false code⟩}
```

This function evaluates the ⟨*test dimension expression*⟩ and compares this in turn to each of the ⟨*dimension expression cases*⟩. If the two are equal then the associated ⟨*code*⟩ is left in the input stream and other cases are discarded. If any of the cases are matched, the ⟨*true code*⟩ is also inserted into the input stream (after the code for the appropriate case), while if none match then the ⟨*false code*⟩ is inserted.

Chapter 10

Comma Separated Lists (`Clist`)

10.1 Constant and Scratch Comma Lists

`\cEmptyClist`

Constant that is always empty.

`\lTmpaClist \lTmpbClist \lTmpcClist \lTmpiClist \lTmpjClist \lTmpkClist`

Scratch comma 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.

`\gTmpaClist \gTmpbClist \gTmpcClist \gTmpiClist \gTmpjClist \gTmpkClist`

Scratch comma 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.

10.2 Creating and Using Comma Lists

`\ClistNew` ⟨*comma list*⟩

Creates a new ⟨*comma list*⟩ or raises an error if the name is already taken. The declaration is global. The ⟨*comma list*⟩ initially contains no items.

`\ClistConst` ⟨*clist var*⟩ `{`⟨*comma list*⟩`}`

Creates a new constant ⟨*clist var*⟩ or raises an error if the name is already taken. The value of the ⟨*clist var*⟩ is set globally to the ⟨*comma list*⟩.

`\ClistVarJoin` ⟨*clist var*⟩ `{`⟨*separator*⟩`}`

Returns the contents of the ⟨*clist var*⟩, with the ⟨*separator*⟩ between the items. For example,

```
\ClistSet \lTmpaClist { a , b , , c , {de} , f }
\ClistVarJoin \lTmpaClist { and }
```

a and b and c and de and f

\ClistVarJoinExtended ⟨*clist var*⟩ {⟨*separator between two*⟩} {⟨*separator between more than two*⟩} {⟨*separator between final two*⟩}

Returns the contents of the ⟨*clist var*⟩, with the appropriate ⟨*separator*⟩ between the items. Namely, if the comma list has more than two items, the ⟨*separator between more than two*⟩ is placed between each pair of items except the last, for which the ⟨*separator between final two*⟩ is used. If the comma list has exactly two items, then they are joined with the ⟨*separator between two*⟩ and returns. For example,

```
\ClistSet \lTmpaClist { a , b , , c , {de} , f }
\ClistVarJoinExtended \lTmpaClist { and } {, } {, and }
```

a, b, c, de, and f

The first separator argument is not used in this case because the comma list has more than 2 items.

\ClistJoin ⟨*comma list*⟩ {⟨*separator*⟩}
\ClistJoinExtended ⟨*comma list*⟩ {⟨*separator between two*⟩} {⟨*separator between more than two*⟩} {⟨*separator between final two*⟩}

Returns the contents of the ⟨*comma list*⟩, with the appropriate ⟨*separator*⟩ between the items. As for **\ClistSet**, blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. The ⟨*separators*⟩ are then inserted in the same way as for **\ClistVarJoin** and **\ClistVarJoinExtended**, respectively.

10.3 Viewing Comma Lists

\ClistLog {⟨*tokens*⟩}

Writes the entries in the comma list in the log file. See also **\ClistShow** which displays the result in the terminal.

\ClistVarLog ⟨*comma list*⟩

Writes the entries in the ⟨*comma list*⟩ in the log file. See also **\ClistVarShow** which displays the result in the terminal.

\ClistShow {⟨*tokens*⟩}

Displays the entries in the comma list in the terminal.

\ClistVarShow ⟨*comma list*⟩

Displays the entries in the ⟨*comma list*⟩ in the terminal.

10.4 Setting Comma Lists

\ClistSet ⟨*comma list*⟩ {⟨$item_1$⟩,...,⟨$item_n$⟩}

Sets ⟨*comma list*⟩ to contain the ⟨*items*⟩, removing any previous content from the variable. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To store some ⟨*tokens*⟩ as a single ⟨*item*⟩ even if the ⟨*tokens*⟩ contain commas or spaces, add a set of braces: **\ClistSet** ⟨*comma list*⟩ { {⟨*tokens*⟩} }.

`\ClistSetEq` ⟨*comma list*$_1$⟩ ⟨*comma list*$_2$⟩

Sets the content of ⟨*comma list*$_1$⟩ equal to that of ⟨*comma list*$_2$⟩. To set a token list variable equal to a comma list variable, use `\TlSetEq`. Conversely, setting a comma list variable to a token list is unadvisable unless one checks space-trimming and related issues.

`\ClistClear` ⟨*comma list*⟩

Clears all items from the ⟨*comma list*⟩.

`\ClistClearNew` ⟨*comma list*⟩

Ensures that the ⟨*comma list*⟩ exists globally by applying `\ClistNew` if necessary, then applies `\ClistClear` to leave the list empty.

`\ClistConcat` ⟨*comma list*$_1$⟩ ⟨*comma list*$_2$⟩ ⟨*comma list*$_3$⟩

Concatenates the content of ⟨*comma list*$_2$⟩ and ⟨*comma list*$_3$⟩ together and saves the result in ⟨*comma list*$_1$⟩. The items in ⟨*comma list*$_2$⟩ are placed at the left side of the new comma list.

`\ClistPutLeft` ⟨*comma list*⟩ `{`⟨*item*$_1$⟩,...,⟨*item*$_n$⟩`}`

Appends the ⟨*items*⟩ to the left of the ⟨*comma list*⟩. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some ⟨*tokens*⟩ as a single ⟨*item*⟩ even if the ⟨*tokens*⟩ contain commas or spaces, add a set of braces: `\ClistPutLeft` ⟨*comma list*⟩ `{ {`⟨*tokens*⟩`} }`.

`\ClistPutRight` ⟨*comma list*⟩ `{`⟨*item*$_1$⟩,...,⟨*item*$_n$⟩`}`

Appends the ⟨*items*⟩ to the right of the ⟨*comma list*⟩. Blank items are omitted, spaces are removed from both sides of each item, then a set of braces is removed if the resulting space-trimmed item is braced. To append some ⟨*tokens*⟩ as a single ⟨*item*⟩ even if the ⟨*tokens*⟩ contain commas or spaces, add a set of braces: `\ClistPutRight` ⟨*comma list*⟩ `{ {`⟨*tokens*⟩`} }`.

10.5 Modifying Comma Lists

While comma lists are normally used as ordered lists, it may be necessary to modify the content. The functions here may be used to update comma lists, while retaining the order of the unaffected entries.

`\ClistRemoveDuplicates` ⟨*comma list*⟩

Removes duplicate items from the ⟨*comma list*⟩, leaving the left most copy of each item in the ⟨*comma list*⟩. The ⟨*item*⟩ comparison takes place on a token basis, as for `\TlIfEqTF`. This function iterates through every item in the ⟨*comma list*⟩ and does a comparison with the ⟨*items*⟩ already checked. It is therefore relatively slow with large comma lists. Furthermore, it may fail if any of the items in the ⟨*comma list*⟩ contains `{`, `}`, or `#` (assuming the usual TeX category codes apply).

`\ClistRemoveAll` ⟨*comma list*⟩ `{`⟨*item*⟩`}`

Removes every occurrence of ⟨*item*⟩ from the ⟨*comma list*⟩. The ⟨*item*⟩ comparison takes place on a token basis, as for `\TlIfEqTF`. The function may fail if the ⟨*item*⟩ contains `{`, `}`, or `#` (assuming the usual TeX category codes apply).

`\ClistReverse` ⟨*comma list*⟩

Reverses the order of items stored in the ⟨*comma list*⟩.

10.6 Working with the Contents of Comma Lists

`\ClistCount` {⟨*comma list*⟩}
`\ClistVarCount` ⟨*comma list*⟩

Returns the number of items in the ⟨*comma list*⟩ as an ⟨*integer denotation*⟩. The total number of items in a ⟨*comma list*⟩ includes those which are duplicates, i.e. every item in a ⟨*comma list*⟩ is counted.

`\ClistItem` {⟨*comma list*⟩} {⟨*integer expression*⟩}

Indexing items in the ⟨*comma list*⟩ from 1 at the top (left), this function evaluates the ⟨*integer expression*⟩ and returns the appropriate item from the comma list. If the ⟨*integer expression*⟩ is negative, indexing occurs from the bottom (right) of the comma list. When the ⟨*integer expression*⟩ is larger than the number of items in the ⟨*comma list*⟩ (as calculated by `\ClistCount`) then the function returns nothing.

`\ClistVarItem` ⟨*comma list*⟩ {⟨*integer expression*⟩}

Indexing items in the ⟨*comma list*⟩ from 1 at the top (left), this function evaluates the ⟨*integer expression*⟩ and returns the appropriate item from the comma list. If the ⟨*integer expression*⟩ is negative, indexing occurs from the bottom (right) of the comma list. When the ⟨*integer expression*⟩ is larger than the number of items in the ⟨*comma list*⟩ (as calculated by `\ClistVarCount`) then the function returns nothing.

`\ClistRandItem` {⟨*comma list*⟩}
`\ClistVarRandItem` ⟨*clist var*⟩

Selects a pseudo-random item of the ⟨*comma list*⟩. If the ⟨*comma list*⟩ has no item, the result is empty.

10.7 Comma Lists as Stacks

Comma lists can be used as stacks, where data is pushed to and popped from the top of the comma list. (The left of a comma list is the top, for performance reasons.) The stack functions for comma lists are not intended to be mixed with the general ordered data functions detailed in the previous section: a comma list should either be used as an ordered data type or as a stack, but not in both ways.

`\ClistGet` ⟨*comma list*⟩ ⟨*token list variable*⟩
`\ClistGetT` ⟨*comma list*⟩ ⟨*token list variable*⟩ ⟨*true code*⟩
`\ClistGetF` ⟨*comma list*⟩ ⟨*token list variable*⟩ ⟨*false code*⟩
`\ClistGetTF` ⟨*comma list*⟩ ⟨*token list variable*⟩ ⟨*true code*⟩ ⟨*false code*⟩

Stores the left-most item from a ⟨*comma list*⟩ in the ⟨*token list variable*⟩ without removing it from the ⟨*comma list*⟩. The ⟨*token list variable*⟩ is assigned locally.

`\ClistPop` ⟨*comma list*⟩ ⟨*token list variable*⟩
`\ClistPopT` ⟨*comma list*⟩ ⟨*token list variable*⟩ {⟨*true code*⟩}
`\ClistPopF` ⟨*comma list*⟩ ⟨*token list variable*⟩ {⟨*false code*⟩}
`\ClistPopTF` ⟨*comma list*⟩ ⟨*token list variable*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Pops the left-most item from a ⟨*comma list*⟩ into the ⟨*token list variable*⟩, i.e. removes the item from the comma list and stores it in the ⟨*token list variable*⟩. The assignment of the ⟨*token list variable*⟩ is local.

If the ⟨*comma list*⟩ is empty, the value of the ⟨*token list variable*⟩ is not defined in this case and should not be relied upon.

\ClistPush ⟨*comma list*⟩ {⟨*items*⟩}

Adds the {⟨*items*⟩} to the top of the ⟨*comma list*⟩. Spaces are removed from both sides of each item as for any `n`-type comma list.

10.8 Mapping over Comma Lists

When the comma list is given explicitly, spaces are trimmed around each item. If the result of trimming spaces is empty, the item is ignored. Otherwise, if the item is surrounded by braces, one set is removed, and the result is passed to the mapped function. Thus, if the comma list that is being mapped is `{a␣,␣{{b}␣},␣,{},␣{c},}` then the arguments passed to the mapped function are '`a`', '`{b}␣`', an empty argument, and '`c`'.

When the comma list is given as a variable, spaces have already been trimmed on input, and items are simply stripped of one set of braces if any. This case is more efficient than using explicit comma lists.

\ClistMapInline {⟨*comma list*⟩} {⟨*inline function*⟩}
\ClistVarMapInline ⟨*comma list*⟩ {⟨*inline function*⟩}

Applies ⟨*inline function*⟩ to every ⟨*item*⟩ stored within the ⟨*comma list*⟩. The ⟨*inline function*⟩ should consist of code which receives the ⟨*item*⟩ as `#1`. The ⟨*items*⟩ are returned from left to right. For example

```
\IgnoreSpacesOn
\TlClear \lTmpaTl
\ClistMapInline {one,two,three} {
  \TlPutRight \lTmpaTl {(#1)}
}
\Result {\TlUse\lTmpaTl}
\IgnoreSpacesOff
```

produces (one)(two)(three).

\ClistMapVariable {⟨*comma list*⟩} ⟨*variable*⟩ {⟨*code*⟩}
\ClistVarMapVariable ⟨*comma list*⟩ ⟨*variable*⟩ {⟨*code*⟩}

Stores each ⟨*item*⟩ of the ⟨*comma list*⟩ in turn in the (token list) ⟨*variable*⟩ and applies the ⟨*code*⟩. The ⟨*code*⟩ will usually make use of the ⟨*variable*⟩, but this is not enforced. The assignments to the ⟨*variable*⟩ are local. Its value after the loop is the last ⟨*item*⟩ in the ⟨*comma list*⟩, or its original value if there were no ⟨*item*⟩. The ⟨*items*⟩ are returned from left to right.

10.9 Comma List Conditionals

\ClistIfExist ⟨*comma list*⟩
\ClistIfExistTF ⟨*comma list*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*comma list*⟩ is currently defined. This does not check that the ⟨*comma list*⟩ really is a comma list.

\ClistIfEmpty {⟨*comma list*⟩}
\ClistIfEmptyTF {⟨*comma list*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

Tests if the ⟨*comma list*⟩ is empty (containing no items). The rules for space trimming are as for other `n`-type comma-list functions, hence the comma list `{ , ,, }` (without outer braces) is empty, while `{ ,{},}` (without outer braces) contains one element, which happens to be empty: the comma-list is not empty.

`\ClistVarIfEmpty` ⟨*comma list*⟩
`\ClistVarIfEmptyTF` ⟨*comma list*⟩ `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if the ⟨*comma list*⟩ is empty (containing no items).

`\ClistIfIn` `{`⟨*comma list*⟩`}` `{`⟨*item*⟩`}`
`\ClistIfInTF` `{`⟨*comma list*⟩`}` `{`⟨*item*⟩`}` `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if the ⟨*item*⟩ is present in the ⟨*comma list*⟩. In the case of an `n`-type ⟨*comma list*⟩, the usual rules of space trimming and brace stripping apply. For example

```
\ClistIfInTF { a , {b}  , {b} , c } {b} {Yes} {No}
```

Yes

`\ClistVarIfIn` ⟨*comma list*⟩ `{`⟨*item*⟩`}`
`\ClistVarIfInTF` ⟨*comma list*⟩ `{`⟨*item*⟩`}` `{`⟨*true code*⟩`}` `{`⟨*false code*⟩`}`

Tests if the ⟨*item*⟩ is present in the ⟨*comma list*⟩. In the case of an `n`-type ⟨*comma list*⟩, the usual rules of space trimming and brace stripping apply.

Chapter 11

The Source Code

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

11.1 Interfaces for Functional Programming (Prg)

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

\RequirePackage{expl3}
\ProvidesExplPackage{functional}{2022-03-19}{2022B}
  {^^JLaTeX2 functional interfaces for LaTeX3 programming layer}

\cs_generate_variant:Nn \iow_log:n { V }
\cs_generate_variant:Nn \tl_log:n { e }
\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} }

%% #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 { [O] ~ \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

\int_new:N \g__fun_nesting_level_int

%% #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
      {
        \int_gincr:N \g__fun_nesting_level_int
        \__fun_evaluate:Nn #1 {#2}
        \int_gdecr:N \g__fun_nesting_level_int
        \__fun_return_result:
      }
  }
\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
      {
        \int_gincr:N \g__fun_nesting_level_int
        \__fun_one_argument_gset:nn { 1 } { ##1 }
        \__fun_evaluate:Nn #1 {#2}
        \int_gdecr:N \g__fun_nesting_level_int
        \__fun_return_result:
      }
  }
\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
      {
        \int_gincr:N \g__fun_nesting_level_int
        \__fun_one_argument_gset:nn { 1 } { ##1 }
        \__fun_one_argument_gset:nn { 2 } { ##2 }
        \__fun_evaluate:Nn #1 {#2}
        \int_gdecr:N \g__fun_nesting_level_int
        \__fun_return_result:
      }
  }
\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
      {
        \int_gincr:N \g__fun_nesting_level_int
```

```
        \__fun_one_argument_gset:nn { 1 } { ##1 }
        \__fun_one_argument_gset:nn { 2 } { ##2 }
        \__fun_one_argument_gset:nn { 3 } { ##3 }
        \__fun_evaluate:Nn #1 {#2}
        \int_gdecr:N \g__fun_nesting_level_int
        \__fun_return_result:
      }
  }
\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
      {
        \int_gincr:N \g__fun_nesting_level_int
        \__fun_one_argument_gset:nn { 1 } { ##1 }
        \__fun_one_argument_gset:nn { 2 } { ##2 }
        \__fun_one_argument_gset:nn { 3 } { ##3 }
        \__fun_one_argument_gset:nn { 4 } { ##4 }
        \__fun_evaluate:Nn #1 {#2}
        \int_gdecr:N \g__fun_nesting_level_int
        \__fun_return_result:
      }
  }
\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
      {
        \int_gincr:N \g__fun_nesting_level_int
        \__fun_one_argument_gset:nn { 1 } { ##1 }
        \__fun_one_argument_gset:nn { 2 } { ##2 }
        \__fun_one_argument_gset:nn { 3 } { ##3 }
        \__fun_one_argument_gset:nn { 4 } { ##4 }
        \__fun_one_argument_gset:nn { 5 } { ##5 }
        \__fun_evaluate:Nn #1 {#2}
        \int_gdecr:N \g__fun_nesting_level_int
        \__fun_return_result:
      }
  }
\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
      {
        \int_gincr:N \g__fun_nesting_level_int
        \__fun_one_argument_gset:nn { 1 } { ##1 }
        \__fun_one_argument_gset:nn { 2 } { ##2 }
        \__fun_one_argument_gset:nn { 3 } { ##3 }
        \__fun_one_argument_gset:nn { 4 } { ##4 }
        \__fun_one_argument_gset:nn { 5 } { ##5 }
        \__fun_one_argument_gset:nn { 6 } { ##6 }
        \__fun_evaluate:Nn #1 {#2}
```

```
        \int_gdecr:N \g__fun_nesting_level_int
        \__fun_return_result:
      }
  }
\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
      {
        \int_gincr:N \g__fun_nesting_level_int
        \__fun_one_argument_gset:nn { 1 } { ##1 }
        \__fun_one_argument_gset:nn { 2 } { ##2 }
        \__fun_one_argument_gset:nn { 3 } { ##3 }
        \__fun_one_argument_gset:nn { 4 } { ##4 }
        \__fun_one_argument_gset:nn { 5 } { ##5 }
        \__fun_one_argument_gset:nn { 6 } { ##6 }
        \__fun_one_argument_gset:nn { 7 } { ##7 }
        \__fun_evaluate:Nn #1 {#2}
        \int_gdecr:N \g__fun_nesting_level_int
        \__fun_return_result:
      }
  }
\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
      {
        \int_gincr:N \g__fun_nesting_level_int
        \__fun_one_argument_gset:nn { 1 } { ##1 }
        \__fun_one_argument_gset:nn { 2 } { ##2 }
        \__fun_one_argument_gset:nn { 3 } { ##3 }
        \__fun_one_argument_gset:nn { 4 } { ##4 }
        \__fun_one_argument_gset:nn { 5 } { ##5 }
        \__fun_one_argument_gset:nn { 6 } { ##6 }
        \__fun_one_argument_gset:nn { 7 } { ##7 }
        \__fun_one_argument_gset:nn { 8 } { ##8 }
        \__fun_evaluate:Nn #1 {#2}
        \int_gdecr:N \g__fun_nesting_level_int
        \__fun_return_result:
      }
  }
\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
      {
        \int_gincr:N \g__fun_nesting_level_int
        \__fun_one_argument_gset:nn { 1 } { ##1 }
        \__fun_one_argument_gset:nn { 2 } { ##2 }
        \__fun_one_argument_gset:nn { 3 } { ##3 }
        \__fun_one_argument_gset:nn { 4 } { ##4 }
        \__fun_one_argument_gset:nn { 5 } { ##5 }
```

```
        \__fun_one_argument_gset:nn { 6 } { ##6 }
        \__fun_one_argument_gset:nn { 7 } { ##7 }
        \__fun_one_argument_gset:nn { 8 } { ##8 }
        \__fun_one_argument_gset:nn { 9 } { ##9 }
        \__fun_evaluate:Nn #1 {#2}
        \int_gdecr:N \g__fun_nesting_level_int
        \__fun_return_result:
      }
  }
\cs_generate_variant:Nn \__fun_new_with_arg_ix:Nnn { NnV }

\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 }
\tl_new:N \l__fun_argument_tl

%% #1: function name; #2: argument specifications
\cs_new_protected:Npn \__fun_evaluate:Nn #1 #2
  {
    \__fun_argtype_index_gzero:
    \__fun_arguments_gclear:
    \tl_map_variable:nNn { #2 } \l__fun_argtype_tl % spaces are ignored
      {
        \__fun_argtype_index_gincr:
        \__fun_one_argument_get:eN { \__fun_argtype_index_use: } \l__fun_argument_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 }
  }

\cs_new_protected:Npn \__fun_evaluate_and_put_argument:N #1
  {
    \cs_if_exist:cTF
      {
        __fun_defined_ \exp_last_unbraced:Ne \cs_to_str:N { \tl_head:N #1 } : w
      }
      {
```

```
        #1
        \__fun_arguments_gput:e { { \exp_not:V \gResultTl } }
      }
      {
        \__fun_arguments_gput:e { { \exp_not:V #1 } }
      }
  }

%% #1: argument number; #2: token lists
\cs_new_protected:Npn \__fun_one_argument_gset:nn #1 #2
  {
    \tl_gset:cn
      { g__fun_one_argument_ \int_use:N \g__fun_nesting_level_int _#1_tl } { #2 }
    %\__fun_one_argument_log:nn { #1 } { set }
  }

%% #1: argument number; #2: variable of token lists
\cs_new_protected:Npn \__fun_one_argument_get:nN #1 #2
  {
    \tl_set_eq:Nc
      #2 { g__fun_one_argument_ \int_use:N \g__fun_nesting_level_int _ #1 _tl }
    %\__fun_one_argument_log:nn { #1 } { get }
  }
\cs_generate_variant:Nn \__fun_one_argument_get:nN { eN }

%% #1: argument number; #2: get or set
\cs_new_protected:Npn \__fun_one_argument_log:nn #1 #2
  {
    \tl_log:e
      {
        #2 ~ level _ \int_use:N \g__fun_nesting_level_int _ arg _ #1 ~ = ~
        \exp_not:v
          { g__fun_one_argument_ \int_use:N \g__fun_nesting_level_int _#1_tl }
      }
  }

\int_new:c  { g__fun_argtype_index_ 1 _int }
\int_new:c  { g__fun_argtype_index_ 2 _int }
\int_new:c  { g__fun_argtype_index_ 3 _int }
\int_new:c  { g__fun_argtype_index_ 4 _int }
\int_new:c  { g__fun_argtype_index_ 5 _int }

\cs_new_protected:Npn \__fun_argtype_index_gzero:
  {
    \int_gzero_new:c
      { g__fun_argtype_index_ \int_use:N \g__fun_nesting_level_int _int }
  }

\cs_new_protected:Npn \__fun_argtype_index_gincr:
  {
    \int_gincr:c
      { g__fun_argtype_index_ \int_use:N \g__fun_nesting_level_int _int }
  }

\cs_new:Npn \__fun_argtype_index_use:
  {
```

```
    \int_use:c { g__fun_argtype_index_ \int_use:N \g__fun_nesting_level_int _int }
  }

\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_set_eq:NN \Break \prg_break:
\cs_set_eq:NN \PrgBreak \prg_break:

\cs_set_eq:NN \BreakDo \prg_break:n
\cs_set_eq:NN \PrgBreakDo \prg_break:n

\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 }
  }

\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 }

\cs_new_protected:Npn \__fun_ignore_spaces_on:
  {
    \ExplSyntaxOn
    \char_set_catcode_math_subscript:N \_
    \char_set_catcode_other:N \:
  }
\cs_set_eq:NN \IgnoreSpacesOn \__fun_ignore_spaces_on:
\cs_set_eq:NN \IgnoreSpacesOff \ExplSyntaxOff
```

11.2 Interfaces for Argument Using (Use)

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

\PrgNewFunction \Value { M }
```

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

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

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

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

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

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

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

11.3 Interfaces for Control Structures (Bool)

```
\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

\PrgNewFunction \BoolNew { M } { \bool_new:N #1 }

\PrgNewFunction \BoolConst { M m } { \bool_const:Nn #1 { #2 } }

\PrgNewFunction \BoolSet { M m } {
  \__fun_do_assignment:Nnn #1
    { \bool_gset:Nn #1 { #2 } } { \bool_set:Nn #1 { #2 } }
}

\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 }
  }
```

```
\PrgNewFunction \BoolSetEq { M M }
  {
    \__fun_do_assignment:Nnn #1
      { \bool_gset_eq:NN #1 #2 } { \bool_set_eq:NN #1 #2 }
  }

\PrgNewFunction \BoolLog { m } { \bool_log:n { #1 } }

\PrgNewFunction \BoolVarLog { M } { \bool_log:N #1 }

\PrgNewFunction \BoolShow { m } { \bool_show:n { #1 } }

\PrgNewFunction \BoolVarShow { M } { \bool_show:N #1 }

\PrgNewConditional \BoolIfExist { M }
  {
    \bool_if_exist:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }

\PrgNewConditional \BoolVarIf { M } { \Result { #1 } }

\PrgNewConditional \BoolVarNot { M }
  {
    \bool_if:NTF #1
      { \Result { \cFalseBool } } { \Result { \cTrueBool } }
  }

\PrgNewConditional \BoolVarAnd { M M }
  {
    \bool_lazy_and:nnTF {#1} {#2}
      { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }

\PrgNewConditional \BoolVarOr { M M }
  {
    \bool_lazy_or:nnTF {#1} {#2}
      { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }

\PrgNewConditional \BoolVarXor { M M }
  {
    \bool_xor:nnTF {#1} {#2}
      { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }
```

11.4 Interfaces for Token Lists (Tl)

```
\tl_set_eq:NN \cEmptyTl \c_empty_tl
\tl_set_eq:NN \cSpaceTl \c_space_tl
\tl_set_eq:NN \cNoValueTl \c_novalue_tl

\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

\PrgNewFunction \TlNew { M } { \tl_new:N #1 }

\PrgNewFunction \TlLog { m } { \tl_log:n { #1 } }

\PrgNewFunction \TlVarLog { M } { \tl_log:N #1 }

\PrgNewFunction \TlShow { m } { \tl_show:n { #1 } }

\PrgNewFunction \TlVarShow { M } { \tl_show:N #1 }

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

\PrgNewFunction \TlToStr { m } { \Expand { \tl_to_str:n { #1 } } }

\PrgNewFunction \TlVarToStr { M } { \Expand { \tl_to_str:N #1 } }

\PrgNewFunction \TlConst { M m } { \tl_const:Nn #1 { #2 } }

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

\PrgNewFunction \TlSetEq { M M }
  {
    \__fun_do_assignment:Nnn #1 { \tl_gset_eq:NN #1 #2 } { \tl_set_eq:NN #1 #2 }
  }

\PrgNewFunction \TlConcat { M M M }
  {
    \__fun_do_assignment:Nnn #1
      { \tl_gconcat:NNN #1 #2 #3 } { \tl_concat:NNN #1 #2 #3 }
  }

\PrgNewFunction \TlClear { M }
  {
    \__fun_do_assignment:Nnn #1 { \tl_gclear:N #1 } { \tl_clear:N #1 }
  }

\PrgNewFunction \TlClearNew { M }
  {
    \__fun_do_assignment:Nnn #1 { \tl_gclear_new:N #1 } { \tl_clear_new:N #1 }
  }

\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 \TlReplaceOnce { M m m }
  {
    \__fun_do_assignment:Nnn #1
      { \tl_greplace_once:Nnn #1 {#2} {#3} } { \tl_replace_once:Nnn #1 {#2} {#3} }
  }

\PrgNewFunction \TlReplaceAll { M m m }
  {
    \__fun_do_assignment:Nnn #1
      { \tl_greplace_all:Nnn #1 {#2} {#3} } { \tl_replace_all:Nnn #1 {#2} {#3} }
  }

\PrgNewFunction \TlRemoveOnce { M m }
  {
    \__fun_do_assignment:Nnn #1
      { \tl_gremove_once:Nn #1 {#2} } { \tl_remove_once:Nn #1 {#2} }
  }

\PrgNewFunction \TlRemoveAll { M m }
  {
    \__fun_do_assignment:Nnn #1
      { \tl_gremove_all:Nn #1 {#2} } { \tl_remove_all:Nn #1 {#2} }
  }

\PrgNewFunction \TlTrimSpaces { m } { \Expand { \tl_trim_spaces:n { #1 } } }

\PrgNewFunction \TlVarTrimSpaces { M }
  {
    \__fun_do_assignment:Nnn #1 { \tl_gtrim_spaces:N #1 } { \tl_trim_spaces:N #1 }
  }

\PrgNewFunction \TlCount { m } { \Expand { \tl_count:n { #1 } } }

\PrgNewFunction \TlVarCount { M } { \Expand { \tl_count:N #1 } }

\PrgNewFunction \TlHead { m } { \Expand { \tl_head:n { #1 } } }

\PrgNewFunction \TlVarHead { M } { \Expand { \tl_head:N #1 } }

\PrgNewFunction \TlTail { m } { \Expand { \tl_tail:n { #1 } } }

\PrgNewFunction \TlVarTail { M } { \Expand { \tl_tail:N #1 } }

\PrgNewFunction \TlItem { m m } { \Expand { \tl_item:nn {#1} {#2} } }

\PrgNewFunction \TlVarItem { M m } { \Expand { \tl_item:Nn #1 {#2} } }

\PrgNewFunction \TlRandItem { m } { \Expand { \tl_rand_item:n {#1} } }
```

```
\PrgNewFunction \TlVarRandItem { M } { \Expand { \tl_rand_item:N #1 } }

\PrgNewFunction \TlVarCase   { M m }     { \tl_case:Nn {#1} {#2} }
\PrgNewFunction \TlVarCaseT  { M m n }   { \tl_case:NnT {#1} {#2} {#3} }
\PrgNewFunction \TlVarCaseF  { M m n }   { \tl_case:NnF {#1} {#2} {#3} }
\PrgNewFunction \TlVarCaseTF { M m n n } { \tl_case:NnTF {#1} {#2} {#3} {#4} }

\PrgNewFunction \TlMapInline { m n }
  {
    \tl_map_inline:nn {#1} {#2}
  }

\PrgNewFunction \TlVarMapInline { M n }
  {
    \tl_map_inline:Nn #1 {#2}
  }

\PrgNewFunction \TlMapVariable { m M n }
  {
    \tl_map_variable:nNn {#1} #2 {#3}
  }

\PrgNewFunction \TlVarMapVariable { M M n }
  {
    \tl_map_variable:NNn #1 #2 {#3}
  }

\PrgNewConditional \TlIfExist { M }
  {
    \tl_if_exist:NTF #1 { \Result { \cTrueBool } }  { \Result { \cFalseBool } }
  }

\PrgNewConditional \TlIfEmpty { m }
  {
    \tl_if_empty:nTF {#1} { \Result { \cTrueBool } }  { \Result { \cFalseBool } }
  }

\PrgNewConditional \TlVarIfEmpty { M }
  {
    \tl_if_empty:NTF #1 { \Result { \cTrueBool } }  { \Result { \cFalseBool } }
  }

\PrgNewConditional \TlIfBlank { m }
  {
    \tl_if_blank:nTF {#1} { \Result { \cTrueBool } }  { \Result { \cFalseBool } }
  }

\PrgNewConditional \TlIfEq { m m }
  {
    \tl_if_eq:nnTF {#1} {#2} { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }

\PrgNewConditional \TlVarIfEq { M M }
  {
    \tl_if_eq:NNTF #1 #2 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
```

```
  }

\PrgNewConditional \TlIfIn { m m }
  {
    \tl_if_in:nnTF {#1} {#2} { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }

\PrgNewConditional \TlVarIfIn { M m }
  {
    \tl_if_in:NnTF #1 {#2} { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }

\PrgNewConditional \TlIfSingle { m }
  {
    \tl_if_single:nTF {#1} { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }

\PrgNewConditional \TlVarIfSingle { M }
  {
    \tl_if_single:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }
```

11.5 Interfaces for Strings (Str)

```
\str_set_eq:NN \cAmpersandStr  \c_ampersand_str
\str_set_eq:NN \cAttignStr     \c_atsign_str
\str_set_eq:NN \cBackslashStr  \c_backslash_str
\str_set_eq:NN \cLeftBraceStr  \c_left_brace_str
\str_set_eq:NN \cRightBraceStr \c_right_brace_str
\str_set_eq:NN \cCircumflexStr \c_circumflex_str
\str_set_eq:NN \cColonStr      \c_colon_str
\str_set_eq:NN \cDollarStr     \c_dollar_str
\str_set_eq:NN \cHashStr       \c_hash_str
\str_set_eq:NN \cPercentStr    \c_percent_str
\str_set_eq:NN \cTildeStr      \c_tilde_str
\str_set_eq:NN \cUnderscoreStr \c_underscore_str
\str_set_eq:NN \cZeroStr       \c_zero_str

\str_new:N \lTmpaStr   \str_new:N \lTmpbStr   \str_new:N \lTmpcStr
\str_new:N \lTmpiStr   \str_new:N \lTmpjStr   \str_new:N \lTmpkStr
\str_new:N \l@Funx@Str \str_new:N \l@Funy@Str \str_new:N \l@Funz@Str

\str_new:N \gTmpaStr   \str_new:N \gTmpbStr   \str_new:N \gTmpcStr
\str_new:N \gTmpiStr   \str_new:N \gTmpjStr   \str_new:N \gTmpkStr
\str_new:N \g@Funx@Str \str_new:N \g@Funy@Str \str_new:N \g@Funz@Str

\PrgNewFunction \StrNew { M } { \str_new:N #1 }

\PrgNewFunction \StrLog { m } { \str_log:n { #1 } }

\PrgNewFunction \StrVarLog { M } { \str_log:N #1 }

\PrgNewFunction \StrShow { m } { \str_show:n { #1 } }
```

```
\PrgNewFunction \StrVarShow { M } { \str_show:N #1 }

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

\PrgNewFunction \StrConst { M m } { \str_const:Nn #1 {#2} }

\PrgNewFunction \StrSet { M m }
  {
    \__fun_do_assignment:Nnn #1 { \str_gset:Nn #1 {#2} } { \str_set:Nn #1 {#2} }
  }

\PrgNewFunction \StrSetEq { M M }
  {
    \__fun_do_assignment:Nnn #1 { \str_gset_eq:NN #1 #2 } { \str_set_eq:NN #1 #2 }
  }

\PrgNewFunction \StrConcat { M M M }
  {
    \__fun_do_assignment:Nnn #1
      { \str_gconcat:NNN #1 #2 #3 } { \str_concat:NNN #1 #2 #3 }
  }

\PrgNewFunction \StrClear { M }
  {
    \__fun_do_assignment:Nnn #1 { \str_gclear:N #1 } { \str_clear:N #1 }
  }

\PrgNewFunction \StrClearNew { M }
  {
    \__fun_do_assignment:Nnn #1 { \str_gclear_new:N #1 } { \str_clear_new:N #1 }
  }

\PrgNewFunction \StrPutLeft { M m }
  {
    \__fun_do_assignment:Nnn #1
      { \str_gput_left:Nn #1 {#2} } { \str_put_left:Nn #1 {#2} }
  }

\PrgNewFunction \StrPutRight { M m }
  {
    \__fun_do_assignment:Nnn #1
      { \str_gput_right:Nn #1 {#2} } { \str_put_right:Nn #1 {#2} }
  }

\PrgNewFunction \StrReplaceOnce { M m m }
  {
    \__fun_do_assignment:Nnn #1
      { \str_greplace_once:Nnn #1 {#2} {#3} } { \str_replace_once:Nnn #1 {#2} {#3} }
  }

\PrgNewFunction \StrReplaceAll { M m m }
  {
    \__fun_do_assignment:Nnn #1
      { \str_greplace_all:Nnn #1 {#2} {#3} } { \str_replace_all:Nnn #1 {#2} {#3} }
  }
```

```
\PrgNewFunction \StrRemoveOnce { M m }
  {
    \__fun_do_assignment:Nnn #1
      { \str_gremove_once:Nn #1 {#2} } { \str_remove_once:Nn #1 {#2} }
  }

\PrgNewFunction \StrRemoveAll { M m }
  {
    \__fun_do_assignment:Nnn #1
      { \str_gremove_all:Nn #1 {#2} } { \str_remove_all:Nn #1 {#2} }
  }

%% Avoid naming confict with xstring package
\cs_if_exist:NF \StrCount
  { \PrgNewFunction \StrCount { m } { \Expand { \str_count:n { #1 } } } }

%% Provide another name for \StrCount function
\PrgNewFunction \StrSize { m } { \Expand { \str_count:n { #1 } } }

\PrgNewFunction \StrVarCount { M } { \Expand { \str_count:N #1 } }

\PrgNewFunction \StrHead { m } { \Expand { \str_head:n { #1 } } }

\PrgNewFunction \StrVarHead { M } { \Expand { \str_head:N #1 } }

\PrgNewFunction \StrTail { m } { \Expand { \str_tail:n { #1 } } }

\PrgNewFunction \StrVarTail { M } { \Expand { \str_tail:N #1 } }

\PrgNewFunction \StrItem { m m } { \Expand { \str_item:nn {#1} {#2} } }

\PrgNewFunction \StrVarItem { M m } { \Expand { \str_item:Nn #1 {#2} } }

\PrgNewFunction \StrCase   { m m }     { \str_case:nn {#1} {#2} }
\PrgNewFunction \StrCaseT  { m m n }   { \str_case:nnT {#1} {#2} {#3} }
\PrgNewFunction \StrCaseF  { m m n }   { \str_case:nnF {#1} {#2} {#3} }
\PrgNewFunction \StrCaseTF { m m n n } { \str_case:nnTF {#1} {#2} {#3} {#4} }

\PrgNewFunction \StrMapInline { m n }
  {
    \str_map_inline:nn {#1} {#2}
  }

\PrgNewFunction \StrVarMapInline { M n }
  {
    \str_map_inline:Nn #1 {#2}
  }

\PrgNewFunction \StrMapVariable { m M n }
  {
    \str_map_variable:nNn {#1} #2 {#3}
  }

\PrgNewFunction \StrVarMapVariable { M M n }
```

```
  {
    \str_map_variable:NNn #1 #2 {#3}
  }

\PrgNewConditional \StrIfExist { M }
  {
    \str_if_exist:NTF #1 { \Result { \cTrueBool } }  { \Result { \cFalseBool } }
  }

\PrgNewConditional \StrVarIfEmpty { M }
  {
    \str_if_empty:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }

\PrgNewConditional \StrIfEq { m m }
  {
    \str_if_eq:nnTF {#1} {#2} { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }

\PrgNewConditional \StrVarIfEq { M M }
  {
    \str_if_eq:NNTF #1 #2 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }

\PrgNewConditional \StrIfIn { m m }
  {
    \str_if_in:nnTF {#1} {#2} { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }

\PrgNewConditional \StrVarIfIn { M m }
  {
    \str_if_in:NnTF #1 {#2} { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }

%% Avoid naming confict with xstring package
\cs_if_exist:NF \StrCompare
  {
    \PrgNewConditional \StrCompare { m N m }
      {
        \str_compare:nNnTF {#1} #2 {#3}
          { \Result { \cTrueBool } }
          { \Result { \cFalseBool } }
      }
  }

%% Provide another name for \StrCompare function
\PrgNewConditional \StrIfCompare { m N m }
  {
    \str_compare:nNnTF {#1} #2 {#3}
      { \Result { \cTrueBool } }
      { \Result { \cFalseBool } }
  }
```

11.6 Interfaces for Integers (Int)

```
\cs_set_eq:NN \cZeroInt        \c_zero_int
```

```
\cs_set_eq:NN \cOneInt         \c_one_int
\cs_set_eq:NN \cMaxInt         \c_max_int
\cs_set_eq:NN \cMaxRegisterInt \c_max_register_int
\cs_set_eq:NN \cMaxCharInt     \c_max_char_in

\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

\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 }
  {
    \Expand { \int_div_round:nn { #1 } { #2 } }
  }

\PrgNewFunction \IntMathDivTruncate { m m }
  {
    \Expand { \int_div_truncate:nn { #1 } { #2 } }
  }

\PrgNewFunction \IntMathSign { m } { \Expand { \int_sign:n { #1 } } }

\PrgNewFunction \IntMathAbs { m } { \Expand { \int_abs:n { #1 } } }

\PrgNewFunction \IntMathMax { m m } { \Expand { \int_max:nn { #1 } { #2 } } }

\PrgNewFunction \IntMathMin { m m } { \Expand { \int_min:nn { #1 } { #2 } } }

\PrgNewFunction \IntMathMod { m m } { \Expand { \int_mod:nn { #1 } { #2 } } }
```

```
\PrgNewFunction \IntMathRand { m m } { \Expand { \int_rand:nn { #1 } { #2 } } }

\PrgNewFunction \IntNew { M } { \int_new:N #1 }

\PrgNewFunction \IntConst { M m } { \int_const:Nn #1 { #2 } }

\PrgNewFunction \IntLog { m } { \int_log:n { #1 } }

\PrgNewFunction \IntVarLog { M } { \int_log:N #1 }

\PrgNewFunction \IntShow { m } { \int_show:n { #1 } }

\PrgNewFunction \IntVarShow { M } { \int_show:N #1 }

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

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

\PrgNewFunction \IntZero { M }
  {
    \__fun_do_assignment:Nnn #1 { \int_gzero:N #1 } { \int_zero:N #1 }
  }

\PrgNewFunction \IntZeroNew { M }
  {
    \__fun_do_assignment:Nnn #1 { \int_gzero_new:N #1 } { \int_zero_new:N #1 }
  }

\PrgNewFunction \IntSetEq { M M }
  {
    \__fun_do_assignment:Nnn #1 { \int_gset_eq:NN #1 #2 } { \int_set_eq:NN #1 #2 }
  }

\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 \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 \IntStepInline { m m m n }
  {
    \int_step_inline:nnnn { #1 } { #2 } { #3 } { #4 }
  }

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

\PrgNewConditional \IntIfExist { M }
  {
    \int_if_exist:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }

\PrgNewConditional \IntIfOdd { m }
  {
    \int_if_odd:nTF { #1 } { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }

\PrgNewConditional \IntIfEven { m }
  {
    \int_if_even:nTF { #1 } { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }

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

\PrgNewFunction \IntCase   { m m }     { \int_case:nn {#1} {#2} }
\PrgNewFunction \IntCaseT  { m m n }   { \int_case:nnT {#1} {#2} {#3} }
\PrgNewFunction \IntCaseF  { m m n }   { \int_case:nnF {#1} {#2} {#3} }
\PrgNewFunction \IntCaseTF { m m n n } { \int_case:nnTF {#1} {#2} {#3} {#4} }
```

11.7 Interfaces for Floating Point Numbers (Fp)

```
\fp_set_eq:NN \cZeroFp      \c_zero_fp
\fp_set_eq:NN \cMinusZeroFp \c_minus_zero_fp
\fp_set_eq:NN \cOneFp       \c_one_fp
\fp_set_eq:NN \cInfFp       \c_inf_fp
\fp_set_eq:NN \cMinusInfFp  \c_minus_inf_fp
\fp_set_eq:NN \cEFp         \c_e_fp
\fp_set_eq:NN \cPiFp        \c_pi_fp
\fp_set_eq:NN \cOneDegreeFp \c_one_degree_fp

\fp_new:N \lTmpaFp   \fp_new:N \lTmpbFp   \fp_new:N \lTmpcFp
\fp_new:N \lTmpiFp   \fp_new:N \lTmpjFp   \fp_new:N \lTmpkFp
\fp_new:N \l@Funx@Fp \fp_new:N \l@Funy@Fp \fp_new:N \l@Funz@Fp

\fp_new:N \gTmpaFp   \fp_new:N \gTmpbFp   \fp_new:N \gTmpcFp
```

```
\fp_new:N \gTmpiFp   \fp_new:N \gTmpjFp   \fp_new:N \gTmpkFp
\fp_new:N \g@Funx@Fp \fp_new:N \g@Funy@Fp \fp_new:N \g@Funz@Fp

\PrgNewFunction \FpEval { m }
  {
    \Result { \Expand { \fp_eval:n { #1 } } }
  }

\PrgNewFunction \FpMathAdd { m m }
  {
    \fp_set:Nn \l@Funx@Fp { \fp_eval:n { (#1) + (#2) } }
    \Result { \FpUse \l@Funx@Fp }
  }

\PrgNewFunction \FpMathSub { m m }
  {
    \fp_set:Nn \l@Funx@Fp { \fp_eval:n { (#1) - (#2) } }
    \Result { \FpUse \l@Funx@Fp }
  }

\PrgNewFunction \FpMathMult { m m }
  {
    \fp_set:Nn \l@Funx@Fp { \fp_eval:n { (#1) * (#2) } }
    \Result { \FpUse \l@Funx@Fp }
  }

\PrgNewFunction \FpMathDiv { m m }
  {
    \fp_set:Nn \l@Funx@Fp { \fp_eval:n { (#1) / (#2) } }
    \Result { \FpUse \l@Funx@Fp }
  }

\PrgNewFunction \FpMathSign { m }
  {
    \Result { \Expand { \fp_sign:n { #1 } } }
  }

\PrgNewFunction \FpMathAbs { m }
  {
    \Result { \Expand { \fp_abs:n { #1 } } }
  }

\PrgNewFunction \FpMathMax { m m }
  {
    \Result { \Expand { \fp_max:nn { #1 } { #2 } } }
  }

\PrgNewFunction \FpMathMin { m m }
  {
    \Result { \Expand { \fp_min:nn { #1 } { #2 } } }
  }

\PrgNewFunction \FpNew { M } { \fp_new:N #1 }

\PrgNewFunction \FpConst { M m } { \fp_const:Nn #1 {#2} }
```

```
\PrgNewFunction \FpUse { M } { \Result { \Expand { \fp_use:N #1 } } }

\PrgNewFunction \FpLog { m } { \fp_log:n { #1 } }

\PrgNewFunction \FpVarLog { M } { \fp_log:N #1 }

\PrgNewFunction \FpShow { m } { \fp_show:n { #1 } }

\PrgNewFunction \FpVarShow { M } { \fp_show:N #1 }

\PrgNewFunction \FpSet { M m }
  {
    \__fun_do_assignment:Nnn #1 { \fp_gset:Nn #1 {#2} } { \fp_set:Nn #1 {#2} }
  }

\PrgNewFunction \FpSetEq { M M }
  {
    \__fun_do_assignment:Nnn #1 { \fp_gset_eq:NN #1 #2 } { \fp_set_eq:NN #1 #2 }
  }

\PrgNewFunction \FpZero { M }
  {
    \__fun_do_assignment:Nnn #1 { \fp_gzero:N #1 } { \fp_zero:N #1 }
  }

\PrgNewFunction \FpZeroNew { M }
  {
    \__fun_do_assignment:Nnn #1 { \fp_gzero_new:N #1 } { \fp_zero_new:N #1 }
  }

\PrgNewFunction \FpAdd { M m }
  {
    \__fun_do_assignment:Nnn #1 { \fp_gadd:Nn #1 {#2} } { \fp_add:Nn #1 {#2} }
  }

\PrgNewFunction \FpSub { M m }
  {
    \__fun_do_assignment:Nnn #1 { \fp_gsub:Nn #1 {#2} } { \fp_sub:Nn #1 {#2} }
  }

\PrgNewFunction \FpStepInline { m m m n }
  {
    \fp_step_inline:nnnn { #1 } { #2 } { #3 } { #4 }
  }

\PrgNewFunction \FpStepVariable { m m m M n }
  {
    \fp_step_variable:nnnNn { #1 } { #2 } { #3 } #4 { #5 }
  }

\PrgNewConditional \FpIfExist { M }
  {
    \fp_if_exist:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }
```

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

11.8 Interfaces for Dimensions (Dim)

```
\cs_set_eq:NN \cMaxDim  \c_max_dim
\cs_set_eq:NN \cZeroDim \c_zero_dim

\dim_new:N \lTmpaDim   \dim_new:N \lTmpbDim   \dim_new:N \lTmpcDim
\dim_new:N \lTmpiDim   \dim_new:N \lTmpjDim   \dim_new:N \lTmpkDim
\dim_new:N \l@Funx@Dim \dim_new:N \l@Funy@Dim \dim_new:N \l@Funz@Dim

\dim_new:N \gTmpaDim   \dim_new:N \gTmpbDim   \dim_new:N \gTmpcDim
\dim_new:N \gTmpiDim   \dim_new:N \gTmpjDim   \dim_new:N \gTmpkDim
\dim_new:N \g@Funx@Dim \dim_new:N \g@Funy@Dim \dim_new:N \g@Funz@Dim

\PrgNewFunction \DimEval { m }
  {
    \Result { \Expand { \dim_eval:n { #1 } } }
  }

\PrgNewFunction \DimMathAdd { m m }
  {
    \dim_set:Nn \l@Funx@Dim { \dim_eval:n { (#1) + (#2) } }
    \Result { \Value \l@Funx@Dim }
  }

\PrgNewFunction \DimMathSub { m m }
  {
    \dim_set:Nn \l@Funx@Dim { \dim_eval:n { (#1) - (#2) } }
    \Result { \Value \l@Funx@Dim }
  }

\PrgNewFunction \DimMathSign { m }
  {
    \Result { \Expand { \dim_sign:n { #1 } } }
  }

\PrgNewFunction \DimMathAbs { m }
  {
    \Result { \Expand { \dim_abs:n { #1 } } }
  }

\PrgNewFunction \DimMathMax { m m }
  {
    \Result { \Expand { \dim_max:nn { #1 } { #2 } } }
  }

\PrgNewFunction \DimMathMin { m m }
  {
```

```
    \Result { \Expand { \dim_min:nn { #1 } { #2 } } }
  }

\PrgNewFunction \DimMathRatio { m m }
  {
    \Result { \Expand { \dim_ratio:nn { #1 } { #2 } } }
  }

\PrgNewFunction \DimNew { M } { \dim_new:N #1 }

\PrgNewFunction \DimConst { M m } { \dim_const:Nn #1 {#2} }

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

\PrgNewFunction \DimLog { m } { \dim_log:n { #1 } }

\PrgNewFunction \DimVarLog { M } { \dim_log:N #1 }

\PrgNewFunction \DimShow { m } { \dim_show:n { #1 } }

\PrgNewFunction \DimVarShow { M } { \dim_show:N #1 }

\PrgNewFunction \DimSet { M m }
  {
    \__fun_do_assignment:Nnn #1 { \dim_gset:Nn #1 {#2} } { \dim_set:Nn #1 {#2} }
  }

\PrgNewFunction \DimSetEq { M M }
  {
    \__fun_do_assignment:Nnn #1 { \dim_gset_eq:NN #1 #2 } { \dim_set_eq:NN #1 #2 }
  }

\PrgNewFunction \DimZero { M }
  {
    \__fun_do_assignment:Nnn #1 { \dim_gzero:N #1 } { \dim_zero:N #1 }
  }

\PrgNewFunction \DimZeroNew { M }
  {
    \__fun_do_assignment:Nnn #1 { \dim_gzero_new:N #1 } { \dim_zero_new:N #1 }
  }

\PrgNewFunction \DimAdd { M m }
  {
    \__fun_do_assignment:Nnn #1 { \dim_gadd:Nn #1 {#2} } { \dim_add:Nn #1 {#2} }
  }

\PrgNewFunction \DimSub { M m }
  {
    \__fun_do_assignment:Nnn #1 { \dim_gsub:Nn #1 {#2} } { \dim_sub:Nn #1 {#2} }
  }

\PrgNewFunction \DimStepInline { m m m n }
  {
```

```
    \dim_step_inline:nnnn { #1 } { #2 } { #3 } { #4 }
  }

\PrgNewFunction \DimStepVariable { m m m M n }
  {
    \dim_step_variable:nnnNn { #1 } { #2 } { #3 } #4 { #5 }
  }

\PrgNewConditional \DimIfExist { M }
  {
    \dim_if_exist:NTF #1 { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }

\PrgNewConditional \DimCompare { m N m }
  {
    \dim_compare:nNnTF {#1} #2 {#3}
      { \Result { \cTrueBool } } { \Result { \cFalseBool } }
  }

\PrgNewFunction \DimCase   { m m }     { \dim_case:nn {#1} {#2} }
\PrgNewFunction \DimCaseT  { m m n }   { \dim_case:nnT {#1} {#2} {#3} }
\PrgNewFunction \DimCaseF  { m m n }   { \dim_case:nnF {#1} {#2} {#3} }
\PrgNewFunction \DimCaseTF { m m n n } { \dim_case:nnTF {#1} {#2} {#3} {#4} }
```

11.9 Interfaces for Sorting Functions (Sort)

```
\cs_set_eq:NN \SortReturnSame \sort_return_same:
\cs_set_eq:NN \SortReturnSwapped \sort_return_swapped:
```

11.10 Interfaces for Comma Separated Lists (Clist)

```
\clist_new:N \lTmpaClist   \clist_new:N \lTmpbClist   \clist_new:N \lTmpcClist
\clist_new:N \lTmpiClist   \clist_new:N \lTmpjClist   \clist_new:N \lTmpkClist
\clist_new:N \l@Funx@Clist \clist_new:N \l@Funy@Clist \clist_new:N \l@Funz@Clist

\clist_new:N \gTmpaClist   \clist_new:N \gTmpbClist   \clist_new:N \gTmpcClist
\clist_new:N \gTmpiClist   \clist_new:N \gTmpjClist   \clist_new:N \gTmpkClist
\clist_new:N \g@Funx@Clist \clist_new:N \g@Funy@Clist \clist_new:N \g@Funz@Clist

\clist_set_eq:NN \cEmptyClist \c_empty_clist

\PrgNewFunction \ClistNew { M } { \clist_new:N #1 }

\PrgNewFunction \ClistLog { m } { \clist_log:n { #1 } }

\PrgNewFunction \ClistVarLog { M } { \clist_log:N #1 }

\PrgNewFunction \ClistShow { m } { \clist_show:n { #1 } }

\PrgNewFunction \ClistVarShow { M } { \clist_show:N #1 }

\PrgNewFunction \ClistVarJoin { M m }
```

```
  {
    \Expand { \clist_use:Nn #1 { #2 } }
  }

\PrgNewFunction \ClistVarJoinExtended { M m m m }
  {
    \Expand { \clist_use:Nnnn #1 { #2 } { #3 } { #4 } }
  }

\PrgNewFunction \ClistJoin { m m }
  {
    \Expand { \clist_use:nn { #1 } { #2 } }
  }

\PrgNewFunction \ClistJoinExtended { m m m m }
  {
    \Expand { \clist_use:nnnn { #1 } { #2 } { #3 } { #4 } }
  }

\PrgNewFunction \ClistConst { M m } { \clist_const:Nn #1 { #2 } }

\PrgNewFunction \ClistSet { M m }
  {
    \__fun_do_assignment:Nnn #1
      { \clist_gset:Nn #1 {#2} } { \clist_set:Nn #1 {#2} }
  }

\PrgNewFunction \ClistSetEq { M M }
  {
    \__fun_do_assignment:Nnn #1
      { \clist_gset_eq:NN #1 #2 } { \clist_set_eq:NN #1 #2 }
  }

\PrgNewFunction \ClistSetFromSeq { M M }
  {
    \__fun_do_assignment:Nnn #1
      { \clist_gset_from_seq:NN #1 #2 } { \clist_set_from_seq:NN #1 #2 }
  }

\PrgNewFunction \ClistConcat { M M M }
  {
    \__fun_do_assignment:Nnn #1
      { \clist_gconcat:NNN #1 #2 #3 } { \clist_concat:NNN #1 #2 #3 }
  }

\PrgNewFunction \ClistClear { M }
  {
    \__fun_do_assignment:Nnn #1 { \clist_gclear:N #1 } { \clist_clear:N #1 }
  }

\PrgNewFunction \ClistClearNew { M }
  {
    \__fun_do_assignment:Nnn #1 { \clist_gclear_new:N #1 } { \clist_clear_new:N #1 }
  }

\PrgNewFunction \ClistPutLeft { M m }
```

```
  {
    \__fun_do_assignment:Nnn #1
      { \clist_gput_left:Nn #1 {#2} } { \clist_put_left:Nn #1 {#2} }
  }

\PrgNewFunction \ClistPutRight { M m }
  {
    \__fun_do_assignment:Nnn #1
      { \clist_gput_right:Nn #1 {#2} } { \clist_put_right:Nn #1 {#2} }
  }

\PrgNewFunction \ClistRemoveDuplicates { M }
  {
    \__fun_do_assignment:Nnn #1
      { \clist_gremove_duplicates:N #1 } { \clist_remove_duplicates:N #1 }
  }

\PrgNewFunction \ClistRemoveAll { M m }
  {
    \__fun_do_assignment:Nnn #1
      { \clist_gremove_all:Nn #1 {#2} } { \clist_remove_all:Nn #1 {#2} }
  }

\PrgNewFunction \ClistReverse { M }
  {
    \__fun_do_assignment:Nnn #1 { \clist_greverse:N #1 } { \clist_reverse:N #1 }
  }

\PrgNewFunction \ClistSort { M m }
  {
    \__fun_do_assignment:Nnn #1
      { \clist_gsort:Nn #1 {#2} } { \clist_sort:Nn #1 {#2} }
  }

\PrgNewFunction \ClistCount { m }
  {
    \Result { \Expand { \clist_count:n { #1 } } }
  }

\PrgNewFunction \ClistVarCount { m }
  {
    \Result { \Expand { \clist_count:N #1 } }
  }

\PrgNewFunction \ClistGet   { M M }     { \clist_get:NN #1 #2 }
\PrgNewFunction \ClistGetT  { M M n }   { \clist_get:NNT #1 #2 {#3} }
\PrgNewFunction \ClistGetF  { M M n }   { \clist_get:NNF #1 #2 {#3} }
\PrgNewFunction \ClistGetTF { M M n n } { \clist_get:NNTF #1 #2 {#3} {#4} }

\PrgNewFunction \ClistPop { M M }
  {
    \__fun_do_assignment:Nnn #1
      { \clist_gpop:NN #1 #2 } { \clist_pop:NN #1 #2 }
  }
\PrgNewFunction \ClistPopT { M M n }
  {
```

```
    \__fun_do_assignment:Nnn #1
      { \clist_gpop:NNT #1 #2 {#3} } { \clist_pop:NNT #1 #2 {#3} }
  }
\PrgNewFunction \ClistPopF { M M n }
  {
    \__fun_do_assignment:Nnn #1
      { \clist_gpop:NNF #1 #2 {#3} } { \clist_pop:NNF #1 #2 {#3} }
  }
\PrgNewFunction \ClistPopTF { M M n n }
  {
    \__fun_do_assignment:Nnn #1
      { \clist_gpop:NNTF #1 #2 {#3} {#4} } { \clist_pop:NNTF #1 #2 {#3} {#4} }
  }

\PrgNewFunction \ClistPush { M m }
  {
    \__fun_do_assignment:Nnn #1
      { \clist_gpush:Nn #1 {#2} } { \clist_push:Nn #1 {#2} }
  }

\PrgNewFunction \ClistItem { m m } { \Expand { \clist_item:nn {#1} {#2} } }

\PrgNewFunction \ClistVarItem { M m } { \Expand { \clist_item:Nn #1 {#2} } }

\PrgNewFunction \ClistRandItem { m } { \Expand { \clist_rand_item:n {#1} } }

\PrgNewFunction \ClistVarRandItem { M } { \Expand { \clist_rand_item:N #1 } }

\PrgNewFunction \ClistMapInline { m n }
  {
    \clist_map_inline:nn {#1} {#2}
  }

\PrgNewFunction \ClistVarMapInline { M n }
  {
    \clist_map_inline:Nn #1 {#2}
  }

\PrgNewFunction \ClistMapVariable { m M n }
  {
    \clist_map_variable:nNn {#1} #2 {#3}
  }

\PrgNewFunction \ClistVarMapVariable { M M n }
  {
    \clist_map_variable:NNn #1 #2 {#3}
  }

\cs_set_eq:NN \ClistMapBreak \clist_map_break:

\PrgNewConditional \ClistIfExist { M }
  {
    \clist_if_exist:NTF #1 { \Result { \cTrueBool } }  { \Result { \cFalseBool } }
  }
```

```
\PrgNewConditional \ClistIfEmpty { m }
  {
    \clist_if_empty:nTF {#1} { \Result { \cTrueBool } }  { \Result { \cFalseBool } }
  }

\PrgNewConditional \ClistVarIfEmpty { m }
  {
    \clist_if_empty:NTF #1 { \Result { \cTrueBool } }  { \Result { \cFalseBool } }
  }

\PrgNewConditional \ClistIfIn { m m }
  {
    \clist_if_in:nnTF {#1} {#2}
      { \Result { \cTrueBool } }  { \Result { \cFalseBool } }
  }

\PrgNewConditional \ClistVarIfIn { M m }
  {
    \clist_if_in:NnTF #1 {#2}
      { \Result { \cTrueBool } }  { \Result { \cFalseBool } }
  }
```