# Template Using Frugal Programming Language (TUF-PL)

Rainer Glaschick, Paderborn
2020-01-28

This is version 0.5 of TUF-PL and its environment.

# Contents

# 1. Introduction

The programming language described in this document was created as a personal exercise to describe

and implement a simple, but versatile programming language, that is easy to write, use and understand, yet more efficient than most interpreted languages. I started with this project in about 1990 (after already 20 years in computer programming) when microprocessors and personal computers came up, in order to have a simple programming language. While Python is close to my programming taste — it avoids the legacy of parenthesis from the paper tape area by using indentation, and dared to provide arbitrary precision integers as default — it still uses the notation of a function with a parameter list, requiring a pair of parenthesis for each call. But it is possible to intermix parameters and words in calling a function[1].

The most prominent features are:

- The core language has no keywords, only symbols (mathematicians, enjoy).
- Functions are named verbosely, intermixing words and parameters (mathematicians, dashed)
- Only seven kinds of items.
- Declaration of variables is not necessary.
- Constraints (under evaluation) create a discretionary type system.
- Parenthesis are sparingly necessary due to indentation awareness and function templates
- Compiled via C, thus efficient and easily provided.
- Arrays grow dynamically.
- Memory management avoids unsolicited garbage collection delays.
- No introspection and self-modification possible.

So the basic syntax is terse, using only symbols, but the functions are strings of words, and rather verbose, with intermixed parameters. As the words and phrases of function templates may undergo localisation, there is no need to learn English terminology; a program can be expressed in any language (assuming that attributes can also be localized).

Note that the language is not strictly object oriented[2], but allows object oriented methodology to be applied fairly easy.

There is no compile-time enforced strict type system; however, the constraint mechanism — if supported by a future compiler — is powerful and probably more flexible.

Thus, it is a language that can be easily used by beginners, but has all necessary features for professional programming; furthermore, it can be fully localised, as there are no reserved words.

The name could be read as an acronym for:

```
Template
Using
Frugal
Programming
Language
```

Originally, it was:

```
Terse
Untyped
Functional
Programming
Language
```

It is terse, because no keywords are used in the core language; all syntax is expressed using special characters and digraphs. although the function strings are long and verbose.

It is untyped, in so far as a variable is not bound to certain type. However, each individual item refers to by a variable is of a certain kind, i.e. number, string, bunch etc. A discretionary type system is designed and hopefully integrates well.

Function templates play a central role, and all parameters are passed uniformly by reference. The term *functional* was mistakenly selected to emphasize the central role of functions; but there are still operators, assignments etc, thus it is not what is claimed a *functional* system. Also, parameters that are rows and arrays with fields can be modified affecting the caller's view, contradicting the functional programming view.

I had observed that nowadays programming depends on libraries, and even fairly basic operations are

done by library calls. Thus, it seemed logical to translate all statements to subroutine calls in C and avoid the overhead of virtual machine code interpretation. Even with modern JIT compilers, early tests with Python and AWK showed that equivalent TUF-PL programs are significantly quicker[3].

Those who think that better words should be used, are free to use the following words as stimuli:

```
tiny tricky totally tested true trash
unique useless useful uncommon unobtrusive unprecedented unpretentious unusual unfill
fancy fast fatless flashy foxy froward fulgent
```

# 2. Language Elements

## 2.1. Lines and Statements

The program consists of a sequence of lines that contain statements. Normally, a statement is just a line.

Statements may be:

function calls:
```
print "Hello world"
```
assignments:
```
x =: 1
```
function returns:
```
:> "Error"
```
guards (if)
```
? i > 0
```
loops (*while*)
```
?* i < 15
```
scans (*for*)
```
?# i =: from 1 upto 15
```
assertions:
```
! i >= 0
```
loop break or repeat:
```
?>
?^
```

Statements — as opposed to the programming language C — cannot be used where an expression is expected, because they don't have a value. This excludes many compact notations, that tend to be complex and hard to understand and verify.

### Lines

Normally, a statement is contained in one line.

Newer compiler versions may automatically detect line continuations, if a line ends in an operator symbol, or parenthesis are still unbalanced.

Instead of long lines, it is recommended to break down lines using extra variables; this will increase never execution time using modern optimizing compilers, and be negligible in other cases, but increases readability and thus reliability. Note that breaking down a complex expression allows to comment the parts individually.

If it cannot be avoided, a join of two lines can be forced by placing the double dot digraph (`..`) as the last symbol of a line (white space including comments may follow and is ignored).

### Statements

The program is composed of statements, normally one on a line.

Statements can be:

- assignments
- guards (conditional and looped blocks) and guard exits
- function call and return
- assertions
- empty statements

As empty lines are ignored and a comment is treated as white space, the empty (do nothing) statement requires a notation to define indentation; a line with just the void notation ( ( ) may be used (or a single semicolon, if condensed notation is available). No good examples have been found so far, as there are no complex loop controls with side effects (and thus it is still not implemented).

First, the line is converted to a row of terms separated by white space, where a term is:

- a word (letter, followed by letters or digits or underscores)
- a number literal
- a string literal
- a symbol

White space must be used if necessary to separate terms, but is optionally otherwise. A symbol may be composed of one or two characters not separated by white space; in the latter case, it is a digraph. (There are no trigraphs etc.) Underscores may be used inside a word. Note that the word must be followed by blank space as usual, if a number literal follows the word.

Then, the line is parsed from left to right and sublines generated as follows:

- An open parenthesis starts recursively a subline, almost always an expression
- A comma separates list components
- White space joins word, identifiers, literals and expressions for function template matching.

Thus, the line is a list of identifiers, constants, symbols or expressions. Then, symbols are processed in priority order; for unary operators, a pair, and for binary operators, a triple with expressions for left hand side and right hand side is set.

Note that the assignment is not an operator; when found as symbol, it tells the line is an assignment. Thus it does not yield a value.

Parsing of the line proceeds as follows:

1. If the last symbol is a colon, it is a function definition template
2. If it starts with an exclamation mark, it is an assertion or starts a constraint
3. If it starts with a question mark or a digraph that starts with a question mark, it is a guard or loop header.

If none of the above is the case, the line is parsed into a tree, i.e. a list of lists. A list is composed of a:

- number constant
- string constant
- word, i.e. a letter followed by letters or digits
- operator symbol (unary or binary, including the double backtick)
- sublist

List elements may be separated either by white space or by commas, where the comma has a higher priority. So a space separated list is a sublist within a comma separated list; no parenthesis are needed.

Opening parenthesis and single backticks start a subexpression that is ended by a (matched) closing parenthesis or a backtick, resp.

An binary operator symbol reorders the list in such a way that it has the operator first, then the left hand sublist, then the right hand sublist.

A unary operator symbol splits the list into the operator symbol and the right hand sublist.

However, the list has first to be scanned if it contains more than one operator, in which case the priority of the operator determines the order in which the list is split.

Once all this has been done, the tree is scanned for sublists starting with a name.

## 2.2. Comments and Pragmas

I use to say that designing a programming language should start with the comment, then the dummy statement that does nothing, and after that the rest may be developed. So comments will be here at an early stage, and pragmas also, as they are syntactically similar to comments.

Comments (text that is completely ignored by the compiler) and pragmas (compiler options) start with a backslash (*&bsl;*) character. Unlike other programming languages, special characters inside strings are not indicated by backslash characters; backslashes inside strings remain backslashes.

Skip to Literals (Constants) when comment and pragmas are boring.

### Comments

Two types of comments are possible:

Block comments:
: A backslash followed by an open parenthesis, i.e. the digraph `\(`, starts the comment, and a backslash followed by a closing parenthesis, i.e. the digraph `\)`, closes it:
`\( this is a block comment \)`

A block comment is often used if the comment block spans several lines; in this case it is strongly recommended to write nothing behind the closing `\)`, as otherwise the indentation might be unclear.

Line tail comments:
: A backslash `\` that is not followed by an open parenthesis or any other character denoting a pragma — thus blank space is preferred — and is not within a block comment treats the rest of the line as comment

Programmers might consider adhering to the following reserved conventions:

- `\-` to denote disabled statements
- `\T` or `\TODO` to denote places to check (TODO)
- `\Q` or `\X` for lines of questionable status, need further inspection; the digraph `\?` may also be used, unless it is in the first column indicating conditional parts.

Some other digraphs are already defined (see next section), but may not be implemented yet:

- `\?` and `\%` for conditional compilation (in column 1)
- `\+`, `\^`, `\$`, `\#` for pragmas (in column 1)
- `\[`, `\]` etc. for alternate comment bracketing

Comments are treated as white space in the source line, i.e. digraphs, strings and identifiers are broken by comments.

To allow integration in shell scripts, if the first line starts with the two characters `#!`, it is treated as comment line.

### Pragmas

For modularisation, there are two pragmas followed by white space and a file path (or URL) :

- `\+` to include the file contents verbatim (Read as *add file contents*).
- `\^` to import the file contents as module, i.e. just scan it for exported function prototypes and named literals. (Read as *look for stuff going upstream outside*). See also Libraries and Modules
- `\$` indicates the (path to) a library that contains binary objects to resolve the function templates imported.

They may be nested; there is no limit imposed by the language. If `\+` is found within a file imported (via `\^`), importing continues, i.e. there is no reverting to include mode.

If the filename is not an absolute one (starting with a slash), the current directory is tried first, i.e. the filename used as is. If not found, the compiler uses a set of include directories to prepend and try (`-I` parameter, `TUFINC` environment variable) until the file is found.

Including a file that is currently open for including would never end, in particular if conditional

compilation is not supported. In order to allow nested includes for libraries that may be redundant, this case is not rejected, but the new file silently ignored. For this, the pathname string is used in comparison, but it may be converted to a real name before comparison on file systems that support soft links.

Include pragmas should in general not be used inside function bodies.

A compiler may support simple conditional compilation; a backslash followed by a question mark (`\?`) in column 1 is used. If followed by a boolean expression in literals, including named constants, the following is considered a comment unless the expression is true. This conditional comment spans upto and including a line that starts with `\?`, with or without a boolean expression. As no expression counts as a true expression, this concludes a conditional expression and is itself a comment line.

It is very often useful to leave debug prints in the programme, but with `debug disable`, the string operations to compose the debug message are still executed and may consume significant time. For cases like this, a short form of conditional compilation uses a single debug flag for the compilation and the `\%` digraph: If the debug compile flag is set, this digraph is ignored; i.e. the rest of the line normally compiled. Otherwise the digraph as well as the rest of the line is ignored. Note that the line becomes a comment, and this may result in indentation errors; in these cases, better use `\? DEBUG` and `\?` to exclude such lines. Also, the conditions `? $DEBUG` and `? is debug enabled` are more efficient at runtime than composing the string argument.

The pragma `\#`, followed by optional blank space and an optional language version pair, may be used to ensure that a compiler for a new version does not accidentally create defective code, e.g. when new attributes are provided. Language version pairs are always a major number, a separator, and a minor number, e.g. `1:12`, `1/12`, `1-12`, etc. Using a point is often used, but it does not indicate a decimal fraction: `1.1` and `1.10` are different versions, while `1.1` would be the same as `1.01`, and `1.3` would come before `1.12` and `1.123`.

After the version pair, a number of keywords may follow to indicate required language features (starting with a plus, read as *required*) and those not used (starting with a minus, read as *not required*). The latter is used to check for backward compatibility when new features are not required. Such keywords may be:

```
Lists
BunchFunctions
FunctionPrototypes
CallbackFunctions
AttributeChangeCallbacks
```

For each major version, the list of available language features is fixed; new language features and thus new keywords require a new major version.

The compiler shall check the version pair and issue a warning if:

- the major version differs
- the minor version given is larger than that of the compiler

A new compiler should be able to compile any older language version correctly, but flag places, where either new features are, perhaps accidentally, used, as well as places where features no longer present in the newest version. If it cannot process that version, the whole programme must be rejected, and not compiled according to some whatever *compatible*[4] version.

Thus, the user can:

- Translate with the old version, and run regression tests. If regression tests fail, this is often an error in the compiler, but may also signal errors in previous compilers.
- Change the code for the old version, still using the old version indication, until all warnings except the one that an old version is compiled, have vanished, and run regression tests.
- Change to the new version number, compile and run regression tests. No errors or warnings should be flagged.
- Apply new features, if desired.

### Debugging

There is a `debug print` function, that immediately returns unless a debug flag is set, and sends its

output to standard error (`stderr`) instead of standard output (`stdout`). No context information is automatically included, the entities `&filename;` and `&lineno;` may be used. There is also:

```
debug print lno (mixed):-   \ print '@&lineno;:' _ mixed
debug print here            \ print '&filename;@&lineno'
debug print stack trace     \ a stack trace
```

Note that `debug print` is dynamically determined; there is no compiler switch to suppress their compilation.

If the expression for a guard can be evaluated at compile time and is false, the body of the guard may not compiled at all:

```
#Debug = ?+     \ or ?- to suppress
...
    ? #Debug = ?+
        print ...       \ executed only if Debug = ?+
    always compiled
```

However, this is not equivalent to a conditional compilation using `\?`, as pragmas inside the block are evaluated.

As TUF-PL is designed and implemented as a precompiler for the C language, it is often desirable for a library to write only the most basic functions in C and the rest in TUF-PL. The canonical way to do so is to provide a short module in C using the TUF-PL interface. If it is ever considered useful to include C code, the digraphs `\{` and `\}` are reserved for this. The easiest way is to copy the lines verbatim into the generated code, which requires intimate knowledge of how code code is generated and mentioned here only as a possible future language extension.

## 2.3. Literals (Constants)

Literals are sometimes called *constants*, although the term *constant* is better reserved for items that are immutable (for a certain time).

Literals allow the creation of items with given contents.

Conceptually, when a literal is assigned to a variable, an item of appropriate kind is created, and a reference to this item stored into the variable.

As plain items are conceptually immutable, assigning a literal to a variable (or bunch element) could be regarded as setting a reference to the respective literal.

The void reference, that indicates that the variable does not contain a valid reference, is completely unique and clearly immutable.

The only item that can be changed is the bunch, which has also features of a structure. To create a new bunch, assign the empty row (`[]`) or array (`{}`) to a variable, and then assign to its cells or fields. See Rows... for details.

### Void literal

The void reference, *void* for short, that refers to nothing, is denoted by a pair of parenthesis as a digraph, i.e. without space within:

```
()
```

Note that `[]` is a new empty row and `{}` a new empty array without elements, and `""` or `''` are empty strings, all not the void reference.

The digraph `()` might be misinterpreted as an empty list; however, there are no empty lists, although `(),)` is a list with only one element, which is a void reference. Note the necessary comma.

From a structural point of view, digraphs like `$.` or `$-` would be a more logical choice, but the empty pair is much better readable. The digraphs `||` or `-|` or `<>` or `~~` had been considered and found to be not clearer.

### Number literals

The basic elements of number literals are sequences of decimal digits, that form non-negative integer numbers which are used to express literals of rational numbers.

Most compilers will evaluate expressions with the usual numerical operators at compile time, thus the minus sign can be used to make a negative integer. If followed by the division operator slash (`/`) and another sequence of digits, a rational number is created `2/3`. Using the exponentiation operator, powers can be expressed, in particular powers of 2 (`2^12`) or 10 (`10^6`). Note that it might be necessary to use parenthesis as in `(2/3)*10^6`. However, integer exponentiation is not simple (see below) and probably not implemented in early compilers.

Floating point (*real*) numbers require a point between two digits, as in `1.1`. As with the exact numbers, additional operators may be used, e.g. in `12.45*10^-6`. The commonly used form `12.45E-6` may be supported due to its compact form, but a decimal point is required (the short form 3E5 is not possible).

Numbers in other bases than 10 must be created from string constants using library functions.

As the comma is a list separator, even if the current locale specifies the comma instead of the decimal point, floating point numbers are created using points between digits. Accepting other locales requires that a pragma is provided to indicate the desired locale, as otherwise a program may not compile in another locale.

In order to have large numbers better readable, in particular in assertions, the apostrophe (single quote, `'`) may be used inside (which does not collide with non-localised strings):

```
n =: 100'000'000
```

Neither blank space nor a comma or point may be used for this purpose.

### String Literals

String literals are rows of characters, enclosed in quotes (`"`) or apostrophes (`'`). Those in quotes are subject to message localization, those in apostrophes not. Character is used in the sense of a Unicode code point, not as a 8-bit byte. Internally, all strings are held as UTF-8 coded strings. If expansion is required, e.g. because heavy modifications of single characters, there is a function to convert a string to a row of code points.

Either type may contain the opposite delimiter; thus, if a string constant for HTML is needed, it is normally not subject to localization, and using the apostrophe is a good choice:

```
print '<a href="'
```

String constants may be joined using the catenation operator, which is a single underscore; an advanced compiler would do this at compile time for string literals. So there is no urgent need to continue strings over line ends, in particular not to include line changes. See below for what is called elsewhere *here documents*.

The underscore character may be used in addition to letters and digits inside words (neither at the begin nor the end); thus the catenation operator is best surrounded by white space; note the difference between `a_b`, which is a single word, and `a _ b`, which results in the catenation of `a` and `b`.

Using a dot sign (enclosed by spaces) looks less readable for me:

```
print 'k=' _ k _ ' c=' _ c
print 'k=' . k . ' c=' . c
```

The plus sign is only useful in strongly typed languages, as it is unclear which one is meant in mixed expressions:

```
k =: 'x+1=' + x + 1     \ is the second plus an addition?
k =: 'x+1=' _ x + 1     \ no longer ambiguous
```

There is no backslash escape for special characters like newlines or other control characters; instead, HTML entities may be used (see https://www.w3schools.com/html/html_entities.asp).

This means, that the string constant *Mühe* results in *Mühe*. Some commonly required named

entities are:

```
quot    "       (double) quote
apos    '       apostrophe
amp     &       ampersand
sect    §       section, paragraph
not     ¬       not
euro    €       Euro currency sign
```

The HTML standard allows any code point using their numerical value, but does not have names for control characters, nor for the backslash, so TUFPL provides additional names for these:

```
nl    &#10; newline
cr    &#13; carriage return
tab   &#9;  tabulator
bsl   &#92; backslash
```

Thus, a text to be localized is written as:

```
"Hab&apos; acht!"     \ ergibt "Hab' acht"!
```

A few HTML entities may in future versions be used as operators, these are `&ne;` for ≠, `&le;` for ≤ and `&ge;` for ≥; the compiler may convert them to unicode or parse them as a single token.

The backslash is conceptually reserved for comments and pragmas, and should be written as `&bsl;` in strings. This makes regular expressions even harder to write as they are anyhow, so there may be abrogated in future versions. Current experience does not indicate a real need inside TUF-PL.

Strings in general support zero bytes, but not in string literals, as these are internally zero terminated strings. Thus, using `&#0;` shall be rejected by the compiler.

If an apostrophe is required in a non-localized string, the HTML entity must be used:

```
print 'mv &apos;' fromfn '&apos; &apos;' tofn '&apos;'
```

Multiline strings (*here documents*) use special digraphs as a starting delimiter for the string. The common form uses `<'` to start a verbatim multiline string, while `<"` starts a localized variant; both are terminated by a single apostrophe or quote as for any other string. HTML entities may be used as with other strings. A leading newline is removed, and the first indentation removed. All following lines have the same indentation removed, if present; tabulators are not expanded here. If after removing the indentation of the last line an empty line results, it is remove also; i.e. a trailing double newline is changed to a single one:

```
s =: <'
    a 1
    b 2
    c 3
    '
! s[1] = 'a' & s[-1] = '&nl;'
```

This string starts with the letter `a` and ends with one newline after the digit `3`.

The alternate multiline string digraphs `>'` and `>"` copy any character verbatim; `s2` will be the same as `s` above:

```
s2 =: >'a 1
b 2
c 3
'
```

Multiline strings can be used to initialise bunches:

```
\ initialisation data as pairs of tags and values:
#Initstr = <'
    1   void
    2   integer
    5   real
```

```
        '
    initialise:
        $initrow =: {}
        ?# lne =: give parts of string #Initstr delimited by newline \ any of '&nl;'
            pair =: split string lne by white space  \ by many of ' &tab;'
            idx =: number from string pair[1]
            $initrow[idx] =: pair[2]
```

Instead of using a string literal, the second line of the function could be written as:

```
        ?# lne =: give parts of string <'
            1   void
            2   integer
            5   real
            ' delimited by newline
```

Another typical example is the *usage* message:

```
    print usage for (progname):
        print replace each '&progname.' by progname in string <"
            Usage: &progname. [options] command [values] '
            Options may be:
                -x  execute
                -v  print version and exit
            "
```

Note that the options are still indented over the first two lines.

Note also that placeholders for dynamic contents must be replaced programmatically as shown, while localisation is done automatically.

String literals are always Unicode strings stored in UTF-8 encoding or pure ASCII strings. If the sourcecode is written in any other encoding, the compiler shall convert it to UTF-8, otherwise the runtime will do so.

If the compiler supports lists, the print statement may accept a list, and the catenation of the strings is done in the print function. As white space cannot be used as list separator on top level, the alternatives are:

```
    print 'x[' _ i _ ']' _ x[i]
    print 'x[' , i , ']' , x[i]
```

And a string with a newline may be written as:

```
    Zeile 1&nl;Zeile2
```

String operations in general create a new string item, but may copy the reference if the result is equal to one of the operands, or if an equal sting already exists. So equal references imply equal contents, but not vice versa. In general, pooling of strings is considered not worth the expense.

Note that the empty string (`""` or `''`) is not the void reference (`()`), but a valid string item that could be used to compare the kind of references.

A special named entity is `&thisline;`, which is the current source line number, and `&callline;`, which is the source line number of the caller. Both are valid also outside strings as integers.

### Named Literals

Not yet implemented

It is often useful to give constant numbers or strings a name to allow consistent use. This is done by named literals.

Named literals begin with the number sign in column 1, followed by a word, followed by an equals sign, and a value, as an expression that can be evaluated at compile time:

```
    #one = 1
```

```
give me two:
    :> #one  + 1
#pi = 3.14159265358979323846
#hello = "Hello"        \ may become "hallo"
```

The expression may contain other named literals; these must be defined lexically before. Otherwise, the location in the source code is arbitrary, but it is strongly recommended to write them before the first use. A compiler option might check this.

The single equals sign is justified because both sides are equal, as in contrast to an assignment.

For the similarity of global variables indicated with a leading dollar, the number sign is used.

Note that the number sign is used otherwise only as part of a digraph. In particular, the conversion to approximated (floating point) numbers is a double number sign, and some special such numbers have digraphs, see Approximate (Floating Point) Numbers.

A number of common constants may be defined automatically; in particular:

```
#pi = 3.14159265358979323846
#piH = #pi / 2.0
#piQ = #pi / 4.0
```

Named literals in a compilation unit (module) that start with a capital letter are considered of outside use and thus are imported by the import (\^) pragma.

The digraph `#!` (hash bang) in the very first line makes the whole line a comment line for use of TUF programs as shell scripts. For the same purpose, a hash followed by blank space may make the line a comment; this may be switched on by `#!` in the first line.

## 2.4. Variables

Variables just hold references to Items, which are similar to objects in other programming languages. Every variable does just hold a reference to an item, or they have none, called the void reference. Bunches like rows and arrays provide a means to save several references to items, and in this sense are aggregates of variables.

Variable names must begin with a letter, followed by letters, digits or underscores, and neither begin nor end with an underscore (so that there is no collision with the string catenation operator). Small and capital letters are distinguished. Using medial capitals (*camel-case*), i.e. capital letters inside words, is preferred over the use of the underscore. However, as long function bodies are discouraged, and function templates allow sequences of words, this is seldom used so far.

Plain variables are local to functions, see below for Global variables.

All variables come into existence if used; there is no declaration, and thus no means to initialise them. To avoid undefined references, each variable is initially set to void (which intentionally is zero, i.e. the NULL pointer in C).

There are no static local variables of functions, thus no need to initialize them before program start. The overhead of dynamic initialisation is negligible as only a relatively small number of variables is concerned (see Bunches (aggregates)).

Variables are not typed; they can hold references to all kinds of items.

Constraint expressions allow a flexible discretionary static type checking, see Assertions and Constraints.

Plain items are conceptually immutable. In the following example, the reference to the number 1 is copied to `b`, so `a` and `b` refer to the same item. However, changing `a` in the third line replaces the reference to the item 1 by a reference to the item 2. There is no means to change the value of a plain item; operators on plain items always deliver a new item.

Example:

```
a =: 1
b =: a
```

```
a =: 2
! b =: 1
```

This is one of the reasons the `i++` notation is not used here; it would look like the item referred to by `i` would be changed.

Although additional information about a plain item can be obtained by using standard functions or attributes of items, these attributes cannot be changed, as plain items are immutable; a new attribute can only be given for a copy thereof.

Consequently, a string may not be e.g. truncated by an operator, only a truncated new string can be generated by a function.

The only mutable kind are bunches (rows and arrays); they can be changed, as they hold references like a set of variables, and the references can be changed. Byte chunks are a third species of bunches, because they are mutable, but do not contain references, just a (large) byte buffer.

In particular, if a reference to a bunch is copied by an assignment, and one reference is used to change a bunch cell or element, then either reference provides exactly the same information, like in this example:

```
r[1] =: 1
r[2] =: 2
p =: r
p[2] =: 3
/ because p and r refer to the same item that was changed:
! p[2] = r[2]
```

This difference coincides with the observation that for plain items comparison is well defined and supported by a predefined operators, while for bunches, there is no canonical way to compare two bunches.

The special variable name `@` is used in some contexts as the current item, often denoted by `this`.

## Global variables

Prefixing the character `$` defines variables which are global to the module (compilation unit). Global variables are thus shared by all functions, but not visible outside the module.

This greatly reduces the implementation effort, they are just `static item* G_name` variables in C. There are neither name collisions with local variables, nor a need to introduce declarations and definitions.

Global variables are not automatically discarded on termination; it is the programmer's obligation to leave all global variables void, otherwise, a memory leak message is likely to occur at termination. While it is a nice side effect to discourage using global variables, the deeper reason is that there are neither automatic module initialisation nor termination functions; like in C, there is just a function with the reserved pattern `main ()` that starts the programme. Using just a few global variables referring to bunches, and using fields of the bunches instead of single global variables, alleviates the burden to clear global variables, as all references within a bunch are automatically freed when the bunch is freed. As callbacks on exit may be registered, this can make it easier:

```
clear globals:
    $glovar =: ()
on normal exit call `clear globals()` with ()
\ or with anonymous functions:
on normal exit call `() $glovar =: ()` with ()
```

In case items that are globally shared between modules are really necessary, there is a reserved global variable `$COMMON` common to all modules, which is initialised with an array and thus maps strings to item references. Using it is done via a key, e.g. in hierarchical notation as in `de.glaschick.tuf`, a hash of a constant string, or via the string representation of a UUID, so that the chance of collision is neglect able. Further protection could be obtained by storing references to bunches with tags. As opposed to normal module-globals, there is no need to remove an entry before termination; this will automatically done on system exit. But of course the item including all its references will stay in memory, until cleared by setting it void.

Another reserved global variable is `$SYSTEM`, which has some special attributes and fields, see below on Fields and Attributes .

It may happen that upon termination of the main programme, that a variable is freed that contains references to millions of items; as these will be diligently freed one per one, so termination may take time, unless the function `exit immediately code (retcode)` is used, that does not check for memory leaks, and thus is highly discouraged. But note that items are freed automatically once there a no longer any references, so this is only the case if all these items were still of potential use.

Note that functions use a different syntax for globality.

## 2.5. Assignments

Normally, an assignment using the assign symbol `=:` has on its left hand side the name of a variable, and on the right hand side an expression that calculates a value, and the variable is assigned a reference to that value. Several other assignment variants are possible, which are all digraphs that start with an equals sign, and all permitted such digraphs are an assignment[5].

The left hand side is called a reference term, or just *term* for short, which allows to store item references, and the right hand side is a value expression, or just *expression* for short, that yields a reference to an item that shall be used to overwrite the reference given by the reference term.

For combined assignment and arithmetic, the digraphs `=+`, `=-`, `=*` and `=/` (only float) are provided, see Arithmetic expressions.

Using `=_`, a string can be appended to the end of a string:

```
a =: x _ y
x =_ y
! a = x

n =: 5
n =_ 'x'
! n = '5x'
```

Note that according to the general rule, a reference to a number can be replaced by a reference to a string, as the above example shows.

A frequent pattern is

```
? x = () :
    x =: something
```

While condensed notation (see Sequences) could be used to squeeze it on one line:

```
? x = () : x =: something
```

the destination has to be written twice identical, which is a (tiny) source of errors, so there is a *set if not void* assignment using the digraph `=~`:

```
x =~ something
```

Another common pattern is to check if a variable refers to a bunch, and to create one, before a field is set:

```
? para = ()
    para =: {}
para.field =: someValue
```

which would be in condensed notation:

```
? para = (): para =: {}; para.field =: someValue
```

and using the *set if not void* assignment:

```
para =~ {}
para.field =: someValue
```

It may be use frequently also to set a tag if not yet set already:

```
item.tag =~ 'aTag'
```

instead of

```
? item.tag = ()
    item.tag =: 'aTag'
```

List assignments allow to assign several values in one assignment, see Lists and list assignments; they are not implemented yet.

## 2.6. Statement Blocks

A block is a sequence of lines with the same or deeper indentation. Its end is reached if the next line has a less deep indentation. A blank line is always ignored and does not end a block, as it would be unclear how many blocks are closed.

Theoretically a block may be empty, if there is no indented line when possible, but there is not yet a use case for it.

The end of the input source file also closes the current block(s).

A line, if it has less indentation as the previous one, must have the same indentation as another line above to match; otherwise, it is an error. A block may be executed conditionally or repeated, if guarded; otherwise, blocks are used for function bodies and other groupings.

Detecting indentation is no problem if only blanks or only tabulators are used for the initial spacing. However, mixed use may be confusing, not only to identify equal depth, but in particular to identify deeper indentation. There are several solutions with different merits:

- Forbid tabs generally
- Allow leading tabs only
- Expand tabs to spaces

The latter requires a tab stop value. Traditional unix uses a raster of 8, i.e at 1, 9, 17, 25, etc, while nowadays with screen editors, a raster of 4 is more convenient (and at the same time the recommended indentation in Python).

Using spaces only is surely the least surprising method; the same is true if indentation is only done by tabulators, as it is independent of the tab stop value.

Despite the experience in Python, tabs are supported by expanding them, with 8 as default tab spacing. *vi* mode lines are detected and may be used to change the default tab spacing for the file in which they occur (at any place, the last one wins). Automatic detection may be feasible, because any dedentation must return to an already used indentation level. However, this has not been tried and is currently not supported. Also, many text editors have this function, and can be used to normalise the use of tabs and spaces.

## 2.7. Guards

A block can be guarded (courtesy to E.W. Dijkstra) by a condition that determines if the block is executed or not.

### Basic guards

A basic guard starts with a question mark (`?`) as first character on the line, followed by a boolean expression. If the expression yields false, the indented (sub-) block that follows the guard command is skipped. If it is true, the indented block is executed.

From a mathematical point of view, no alternative (*else*) is necessary, as the condition could be inverted, as in:

```
? i < 0
    m =: "less"
```

```
? ~(i < 0)
    m =: "greater or equal"
```

However, this puristic approach is not sensible. In particular, as the condition must be written twice the same, which could be error prone.

Thus, the common *otherwise* or *else* is possible, telling that the condition of the immediately proceeding guard is inverted:

```
? i < 0
    m =: "less"
?~
    m =: "greater or equal"
```

As shown, the digraph ?~ would be the logical choice; but using the vertical bar | is much better readable.

The common if-then-else like

```
if (i < 0) {
    m = "less";
} else if (i > 0) {
        m = "greater";
    } else {
        m = "equal";
    }
}
```

becomes:

```
? i < 0
    m =: "less"
|
    ? i > 0
        m =: "greater"
    |
        m =: "equal"
```

The *else* token is the first and only token on a line, not at least to define the indentation of the following block.

Consequently, the combined *elif* in some languages is straightforward, but may not yet be supported by your compiler:

```
? i < 0
    m =: "less"
|? i > 0
    m =: "greater"
|
    m =: "equal"
```

Note that |? is a digraph, and no blank space may be used in between.

Within a guarded block, the block can be terminated or repeated using the break operator ?> or the repeat operator ?^; in the latter case, the guard expression is evaluated again, etc. Both symbols are digraphs, i.e. no white space possible inside.

Thus, to print squares, the program may read:

```
i =: 1
? i < 10
    print i*i
    i =+ 1
    ?^
```

### Repeat loops

For convenience and better readability and because there are so many loops to repeat, the loop guard (?*) implies a repeat at the end of the block, which is essentially the same as the previous example:

```
i =: 1
?* i <10
    print i*i
    i = +1
```

Endless loops have no expression (or the true symbol ?+) after the loop guard (the following example should be replaced by a better one):

```
i =: 1
?*
    ? i > 9
        ?>
    print i*i
    i =+ 1
```

Note that, unlike several other programming languages, an assignment is not an expression; also the start value must be set on an extra line, unless scans like those in the next section are used.

Instead of a boolean expression, values with boolean kind can be used. Just the void reference is allowed alternatively for false, to make array processing easier. Anything else is a fatal runtime error, as the complexity of the rules don't make the programs more reliable or understandable.

Note that there is no rule to fixed indentations, i.e. in:

```
? cond1
    then1
|
        else1
```

the indentation is just bad programming style, but not invalid.

**Block repeat and terminate**

Within a block, the current iteration can be repeated or terminated by the operators ?^ (continue or repeat, go up to loop begin) and ?> (break). It was tempting to systematically use ?< for repeat, but experience showed that when looking for errors, ?< and ?> are not easily distinguished. Thus we have:

```
x =: 1.0
?*
  z =: (a/x - x) / 2.0
  ? z < epsilon
     ?>
  x =+ z
```

As the break digraph is the only one on a line, a multi-level break could be indicated by having two or more in a line:

```
?*                  \ first level loop
    ...
    ?*              \ second level loop
        ...
        ?>          \ exit inner loop, and repeat outer loop
        ...
        ?> ?>       \ exit both loops
```

This is not yet implemented, and if the same is useful for repeats, had not been evaluated.

**Scan guards**

A standard guarded repeat like this:

```
i =: 1
?* i < 10
```

```
    print i*i
    i =+ 1
```

is a bit verbose and error prone because the loop index is to be repeated.

A scan guard allows to write the same loop as:

```
?# i =: from 1 upto 10
    print i*i
```

An assignment with a function at the right hand side is used instead of a boolean expression, and scan functions with specific side effects control the scan as follows:

A special context variable, denoted as `$#` of `$@`, is saved, set to void, and the assignment executed[6].

Any function, called directly or indirectly during the evaluation of the right hand side of the expression, may get and set the value of the context variable using the symbol `$#`. If the context variable is void after the assignment has been done, then the block is not executed; otherwise, it is executed.

The returned value is assigned to the variable in any case, which is needed for fault items that terminate a loop prematurely. It depends on the scan function if it returns void or the next value as the basic scan `from () upto ()` does.

As a void context variable terminates a loop, if the right hand side of the assignment does not set it to something not void, the scan guard block is never executed.

The special variable `$#` is saved and restored to a hidden local variable each time a scan is started and terminated. So scans can be nested to any depth allowed by the stack. Also, the function called at the right hand side of the assignment needs not use the scan variable by itself; it can call a scan function directly or indirectly. Any examples for this have not been found, but this may come with time.

Scan functions often start with the verb *give*, like in `give lines from file (filename)`, or with *from* as the standard integer iterators, etc. There are not yet defined any constraints to ensure that a function is a scan function.

Using the context variable, the standard (integer) loop enumerating integer numbers can be written as follows:

```
from (first) upto (last) step (step):
    ? $# = ()           \ first call?
        $# =: first     \ yes, set context variable
    |
        $# =+ step      \ else advance
    rv =: $#            \ return value
    ? $# > last
        $# =: ()        \ done, clear context variable
    :> rv
```

The parameter expression(s) of a scan function are evaluated each time the loop is repeated; so for the above example, the step and last value may be changed dynamically. This is, however, considered at least dangerous programming and not recommended. It is best to assign all expressions to (temporary) variables and use literals and simple expressions only.

Note that leaving a scan loop by a return or break can be done without any problems; but the the scan function will never know about it; its whole context is in the scan context `$#` anyhow. Global variables in scan functions have not yet been used and are not expected to be coming soon.

If the loop is terminated, the context variable has been set void and all data memory released (if there are no other references).

Because a scan terminates once the scan variable `$#` is void, an assignment of a value not from a scan function never executes the dependent block, because `$#` is unchanged and remains void. The following code that tries to find the last x in s (note the empty body) just returns the value delivered by the first call:

```
i =: 0
?# i =: locate 'x' in 'abcx123x908' from i+1
```

```
! i = 4         \ result of first call
```

There are wrapper scan functions as the following ones:

```
until void (val):
    $# =: val
    :> val

until zero (val):
    $# =: val
    ? val ?= 0          \ not equal if not a number
        $# =: ()
    :> val
```

that allow to write it compact and still clear:

```
scr =: 'abcx123x908'
i =: 1
?# i =: until zero locate 'x' in scr from i+1
    res =: i
print res
```

Do not use a scan function as argument for these wrapper scans; they will endlessly deliver the first result, as the wrapper functions overwrites the setting of `$#`.

Note that a scan function may use as scan guard without problems, but not call a scan function directly. Although this could be detected, it is much to seldom the case to justify the effort.

Of course, this implies that in a scan, all functions calls in the scan function are re-evaluated in each iteration; normally it is best to use only variables, like in

```
occs =: determine occurrences of ...
?# i =: from 1 upto occs
    ....
```

whereas in:

```
?# i =: from 1 upto determine occurrences of ...
```

the function `determine occurrences of ...` is called each time the loop is repeated, which might well be intended.

While the above scans functions are part of the standard library, the mechanism allows to enumerate more unusual sequences, e.g. Fibonacci numbers. A bunch is used to hold two values in the context variable, which is temporarily in `lc` for better readability:

```
enumerate Fibonacci numbers upto (n):-
    lc =: $#
    ? lc = ()
        lc =: {}
        lc.va =: 0
        lc.vb =: 1
    b =: lc.vb
    lc.vb =+ lc.va
    lc.va =: b
    ? b > n
        lc =: ()
    $# =: lc
    :> b
```

Another unusual pattern is shown by the library scans `until void ()` and `until zero ()`, that wrap any function as scan functions. In the following example, the parameter of the wrapped function is modified for each call:

```
expand tabs in string (line):
    ?#  i =: until zero locate string '&tab;' in line
        r =: 1 + $tabset - (i %% $tabset)
```

```
            line =: replace character at i in line by repeat r times string ' '
        :> line
```

Admittedly, the digraph `$#` is ugly, and was chosen to be close to the scan starter `?#`, that was choose to remind of a number of values to be supplied. So, in many examples, the loop context is saved in a plain variable with a name better to read, in particular `lc` as above.

Instead of `$#`, the digraph `$@` is be allowed, like in:

```
enumerate Fibonacci numbers upto (n):+
    ? $@ = ()
        $@ =: {}
        $@.va =: 0
        $@.vb =: 1
    b =: $@.vb
    $@.vb =+ $@.va
    $@.va =: b
    ? b > n
        $
```

the result is only slightly better readable, but one may get accustomed to it.

Using the variable name `this`, the example becomes:

```
enumerate Fibonacci numbers upto (n):+
    this =: $#
    ? this = ()
        this =: {}
        this.va =: 0
        this.vb =: 1
    b =: this.vb
    this.vb =+ this.va
    this.va =: b
    ? b > n
        this =: ()
    $# =: this
    :> b
```

leading to the observation that scan functions could be considered in an object oriented view as bunch functions of a predefined array named `scan` (see Tagged bunches...), as in:

```
scan:: enumerate Fibonacci numbers upto (n):+
    ? @ = ()
        @ =: {}
        @.va =: 0
        @.vb =: 1
    b =: @.vb
    @.vb =+ @.va
    @.va =: b
    ? b > n
        @ =: ()
    :> b
```

Although this allows to mark scan functions clearly — a feature still definitely missing — bunch functions require an already existing tagged bunch, while scan functions start with a void context. So more pondering is necessary, and the scroll symbol alone will not yet be introduced as an alternative for `$@` and `$#`.

**Index scans**

To print all elements of a row, including those that are void, the indices can just be generated in the common way:

```
r =: []
r[1] =: 1
?# i =: from r.first upto r.last
```

```
    print r[i]
```

As this is not possible for Arrays... with arbitrary indices, scan functions are provided, that deliver all indices for non-void elements:

```
indices of row ()               \ indices of non-void elements (ascending)
indices of array ()             \ indices in undefined order
```

As arrays never have void elements, the scan can only deliver the indices for which the elements are not void; for consistency, the row scan does the same. Otherwise, the above `from r.first upto r.last` must be used.

Both are also available as attribute `.scan`.

Undefined order for array indices means the order can be different for each invocation, even if there is no change or access.

To avoid data corruption, it is a fatal error to change the bunch while the above scan functions are in use; for this purpose, the `.revisions` attribute is checked:

```
indices of row (r):
    ? $@ = ()                       \ initialise?
        $@ =: {}
        $@.revno =: r.revs          \ cannot use $@.revs (attribute)
        $@.index =: r.first
    |
        $@.index =+ 1
        ! $@.revno = r.revs         \ must be unchanged
    \ search next non-void element
    ?*
        ? $@.index > r.last         \ end reached ?
            $@ =: ()
            :> ()
        ? r[$@.index] ~= ()         \ void element?
            ?>                      \ no, exit loop
        $@.index =+ 1               \ yes, advance
    :> $@.index
```

Note that providing a different row reference in subsequent calls gives undefined results, although the revisions check will sometimes detect the error.

To provide indices in ascending or descending order for arrays, all indices must be of the same comparable kind, i.e. numbers or strings; otherwise a fatal runtime error occurs. As the set of indices is determined first, the array is not protected against changes.

The number of indices is `a.last - a.first + 1`. If `a.last < a.first`, the row is empty; in particular:

```
row =: []
! row.first = 0
! row.last = -1
row[1] =: 1
! row.first = 1
! row.last = 1
```

There are no non-void cells before `x.first` and after `x.last`, and the enumeration:

```
?# i =: r.scan
    show r[i]
```

does not provide indices for empty cells; if this is desired, use:

```
?# i =: from r.first upto r.last
    show r[i]
```

which may yield any number of indices for which `r[i]` is void, including all, even if there are a lot of indices given.

Note the assertions in:

```
k =: 0
?# i =: from r.first upto r.last
        k =+ 1
! k = r.last - r.first + 1
k =: 0
?# i =: r.scan
    k =+ 1
! k = r.count
```

Initially, before any integer index is used, we have:

```
r =: []
! r.count = 0
! r.first = 0
! r.last = -1
```

The sorted indices can be obtained and used in a scan function:

```
row of indices of array (ary) ascending:
    r =: [ary.count]
    ?# s =: ary.scan
        r[] =: ary[s]       \ r[] = r[1+r.last]
    sort row r inplace
    :> r

indices of (ary) ascending:
    \ if first call, then get the indices as sorted row
    indices =: $#
    ? indices = ()
        indices =: row of indices of array (ary) ascending
        indices.nexti =: 1      \ field to keep track
        $# =: indices
    \ if exhausted, done
    ? indices.nexti > indices.last
        $# =: ()
        :> ()
    \ else deliver next
    rv =: indices[indices.nexti]
    indices.nexti =+ 1
    \ no need to set $#, the bunch field keeps track
    :> rv
```

Note that setting a string indexed cell to void removes it and decreases the `.count` attribute; enumerating string indices will always return indices for non-void values.

While it is simple to traverse a row without a scan function, this is not the case for arrays. If only the value are to be changed, the above generator can be used in an unsorted version:

```
give indices of array (ary) unsorted:
    \ if first call, then sort the array
    indices =: $#
    ? indices = ()
        indices =: [ary.count]
        \ get the keys unsorted
        ?# key =: ary.scan
            indices[] =: key
        indices.nexti =: 1
        $# =: indices
    \ if exhausted, done
    ? indices.nexti > indices.last
        $# =: ()
        :> ()
    \ else deliver next
    rv =: indices[indices.nexti]
```

```
indices.nexti =+ 1
\ no need to set $#, the bunch attribute keeps track
:> rv
```

The row and array iterators do not lock the bunch, but check that the revision number remains the same. Thus, if the bunch is changed during an enumeration a run time error results. An example is an in-place sort, where elements are exchanged during a walk, that can thus not use `.scan`.

Setting an element before `.first` or after `.last` will update these numbers, but not vice versa:

```
r =: []
r[5] =: 5
! r.first = 5, r.last = 5
r[1] =: 1
! r.first = 1, r.last = 5
r[1] =: ()
! r.first = 1
r[5] =: ()
\ no non-void member left, yet:
! r.first = 1, r.last = 5
```

To use a row as a stack, use a field to keep the stack pointer.

The language intentionally does not provide means to use arbitrary strings as keys for fields, as arrays are the proper solution in these cases.

However, for special purposes like serialisation of bunches, a pair of basic library functions allows to obtain the values of (non-void) fields as an array, and to set fields of a bunch from such an array. This pair of functions is sufficient for the special cases; a triple to enumerate, get and set fields individually is considered neither necessary nor useful, in order to discourage the use of fields where a reference to an array would be the right solution:

```
obtain fields from (bunch):
change fields of (bunch) from (array):
```

It would be a consequential to use arrays not only with strings as indices, but with any item except the void value. However, this would only make sense if the items used as indices are `comparable equal`, in order to allow to search for the index. Note that many items for a single number may exist, and that likewise strings could be compared for equality without hassle. While currently the equality comparison refuses to compare items, this might be extended as follows: If an item has a non-void attribute with the name `equal`, which is a function reference, the equal operator will call this function with two parameters, the two indices to compare, and regard the index found if they are equal under as reported by this function. See also under named bunches. Note that named bunches (see below) are nearly indispensable here, as they share all fields, hence the equal attribute too.

## 2.8. Operators

Parenthesis may be used as usual to group subexpressions or lists (they do not create lists, see below).

The comma (`,`) is used as a list element separator, and white space to separate sequence elements.

Note that in complex symbols composed of more than one special character these must follow without intervening spaces.

Execution operators:

```
?       Guard (if)
|?      Guard alternative (elif)
|       Guard default (else)
?*      Loop
?#      Scan (enumerations, iterations)
??      Error Guard
?|      Error catch
=:      Assignment
$       Global variable name prefix
#       Named literal (global) prefix
```

```
!       assertion or constraint
:>      return function value
?^      loop repeat
?>      loop break
<~ ~>   coroutine yield or receive
:       functions
:(      reference call
::      bunch function call
%       list catenation
```

Instead of special characters, (reserved) words may be used:

```
?*  while
?   if
|?  elif
|   else
?*  loop
?#  scan
?^  continue
?>  break
??  try
?|  catch
!   assert
:>  return
<-  yield
```

Note that the character for guard alternative and guard default are the first one on a line and thus cannot be confused with a logical operator.

### Arithmetic expressions

Arithmetic operators are in ascending priority:

```
+   addition
-   subtraction
*   multiplication
/   division for approximated (floating point) or integer numbers for rational result
//  division for accurate numbers (integer and rational, (see [[Integral Numbers]] fo
%%  integer modulus (see [[Integral Numbers]] for variants
^   exponentiation
-   (unary) sign change
##  (unary) convert to floating point (also a attribute and function)
```

See below for item attributes or functions to obtain the absolute value or the sign of numeric item.

The operators accept either exact or floating point numbers, not mixed. To combine exact and floating point numbers, the exact number has to be converted to floating point using the ## operator, the `.float` attribute or the `float()` function. Note that the .float attribute cannot applied to integer literals.

Exponentiation with integer or rational exponents is currently not supported, although mathematically easy. A simple solution for integer exponents would use repeated multiplication, but could easily result in heavy cpu load and unexpected time delays. The solution is well known (take the exponent as binary, and square and multiply), but has not yet had priority for implementation. Exponentiation with rationals provides often irrational numbers, thus the result must be a a floating point number for consistency; so it is better to have the exponent converted explicitly first.

Ideally, a void operand in an arithmetic expression would be a fatal runtime error. However, when counting objects in an array, under this regime the coding has to be e.g.

```
? array{idx} ~= ()
    array{idx} =+ 1
```

so that the array is searched always one more time just to find out if the indexed value is void (even if the last used index is cached). It seems not too dangerous to treat void references as the number 0 in addition and subtraction, as an added zero is neutral. However, in multiplications a zero operand is

dominant, so a void reference is not treated as zero, as in all other arithmetic operations. Adding two voids is a zero integer, so that an arithmetic operation always returns a number, as is the negation, and the conversion of a void reference to floating point numbers.

The consequence is that conversions from integer to string must return a fault item, although a void would be sufficient to indicate that the conversion failed. But this may lead to hard to detect errors, if bad number stings are just treated as zero, because comparing the result to void has be forgotten. So there are two variants:

```
integer from string (x)           \ returns fault item if faulty
integer from string (x) else (y)  \ if not an integer, return y
```

The programmer may still use a void as default, as in:

```
x =: integer from string str else ()
```

to return void on errors, and an integer number otherwise.

The combined assignment and calculation is defined for these:

```
=+  Add rhs to lhs
=-  Subtract rhs from lhs
=*  Multiply lhs by rhs (any number kind)
=/  Divide lhs by rhs (only floating point numbers)
```

The symbols are intentionally inverted over the form known from the C language. Note that all assignments are digraphs starting with an equals sign, while all comparisons that concern include equality end with an equals sign.

There is no equivalent to the famous ++x or x++ construct, which provides the value of x with a side effect; the increment (or decrement) assignment `x =+ 1` must be used as a statement of its own; it also allows any step. Clarity of programming seems to be better this way.

The shortcut =^ for exponentiation is not supported, as it is assumed that it is not used often enough. Also, the shortcut =% for *divide by rhs and supply the remainder* is not allowed, due to the three different version of remainder calculation; consequently, the =/ operator is limited to floating point numbers.

The shortcuts =& and =| are reserved for boolean assignments, even if not yet implemented by most compilers.

Note that no arithmetic operator starts with an equals sign, while all assignments do so.

### Boolean expressions

Very similar to arithmetic expressions are boolean expressions, using these comparison operators:

```
=   equal
~=  not equal ( 'a ~= b' ≡ '~(a = b)' )
<   less
>   greater
<=  not greater ≡ less or equal (see below)
>=  not less ≡ greater or equal (see below)
.=  of the same kind
~.  not of the same kind
?=  guarded equal (of the same kind and equal)
~?  guarded unequal ( a ~? b ≡ ~(a ?= b)
@=  equal tag
~@  unequal tag
```

Note that the tilde is always prepended the symbol that, when followed by an equals sign, gives the comparison that is negated, while assignments always start with an equals sign.

There are no overlaps with arithmetic operators, logical operators and assignment statements. (An overlap with the assignment digraphs is anyhow not possible, as the target of an assignment is not an expression.)

The comparison operator = (*equal*) first checks if the references are equal; if they are (also if both are void), the operator yields true without further inspection of the items.

If one is void and the other one not, it yields false in any case, so any reference can be compared to the void reference with any operator.

If both operands are of the same kind, only numbers, strings or booleans are compared by their value.

Comparing integer with integer, rational with rational and float with float numbers is straightforward; if mixed, we have:

- Integers may be compared with rationals are never equal, as rationals always have a denominator greater than one.
- Order comparison of rationals with rationals or integers is possible, although it is not trivial as concerns overflows..
- Equality comparison of floating point numbers is not supported, as they are approximate numbers and equality comparison is normally negligent.
- Order comparison of floating point numbers supports all four comparisons, even if equality is included, see below.
- Order comparison of floating point numbers with integer or rational numbers is mathematically and technically no problem; but as mixing in arithmetic expressions is not allowed, it is also not allowed unless the programmer converts the other to floating point first, to apply the rules uniformly.

It might be surprising that equality is banned for floating point numbers, but the tests >= and <= are allowed, so that `x <= y & x >= y` could be used to replace the restriction, and clearly indicating a rather suspicious case. The reserved global variable `$SYSTEM` provides the field `$SYSTEM.epsilon`, which is the smallest difference between `1.0` and the next number, so that you might to terminate an iteration by

```
ep =: (a - x) / a
? ep < $SYSTEM.epsilon
    ?>
```

Note that the order comparisons <, >, <= and >= can be done with only one of them, plus negation:

```
a<b ≡ b>a
a<=b ≡ ~(a>b)
a>=b ≡ ~(a<b) ≡ ~(b>a)
```

So all four are done using > (*greater*) only; although the digraph suggests it, no test for equality is done for >= and <=. And the above comparison is equivalent to '~(y > x | x > y) , using only the *greater than* comparison.

Item references of other kinds than integer, rational and approximated (float) require that same kind of both operands; otherwise a fatal runtime error occurs.

Equality comparison of references of the same kind are allowed and check the reference only; no contents is compared.

The fact that comparisons can crash the program might be regarded unfortunate, as one solution might be to let the comparison be false in these cases, and just continue the program instead of aborting it — and provoking all kind of subtle errors later, in particular when the tests are run and the program is distributed.

The *equal kind* (`.=`) comparison allows to use a prototype, as in:

```
x.field .= ''
```

This would otherwise require stacked field names and fields of literals, which are not supported by all compilers:

```
x.field.kind = ''.kind
```

There are cases when two references should be compared if of the same kind, an yielding false otherwise, which would be programmed like:

```
? x.kind ~= y.kind      \ same as x .= y
    :> ?-
? x = y
    :> ?+
|
    :> ?-
```

In these cases, the guarded equality `?=` may be used, that first checks if the kinds are equal; if not, the result is false. Otherwise, the references of same kind are compared.

Admittedly the double question mark in

```
? x ?= 0
```

is a bit hard to read, but guarded comparisons are not quite often needed.

As the attribute notation might be less clear, functions are provided to determine the kind of an item:

```
is () null:
is () integer:
is () rational:
is () float:
is () exact:        \ integer or rational
is () numeric:      \ integer, rational or float
is () string:
is () bunch:        \ array or row
is () row:
is () chunk:
is () fault:
is () faulty:
```

While `is () string` sounds acceptable over the better `is () a string`, the uniform version `is () fault` felt so wrong even for my ears, that `is () faulty` is an alias to `is () fault`, even if I prefer still the attribute notation `x.isfault`

Similar (but without functions yet) is the *equal tag* (`@=`) comparison, which is often used in assertions and constraints.

There is no operator to just check if two references are equal; comparing e.g. two integer references this way is assumed to lead to tricky programming and unreliable results.

There is a long standing discussion whether to use the single equals sign for assignment or for comparison. The extended diagnostics in modern C compilers as well as my personal experience in programming C leads to the conclusion that numerous hours have been wasted in C programming and debugging, and not at least having programmed in ALGOL 60 and PASCAL, the equals sign is for equality test, and the assignment uses the mirrored ALGOL colon-equals (`=:`) for consistency as assignment symbol.

It might be possible to relax this strict rule. The single equals could be used for both, assignment and comparison, because the assignment is not an operator that yields a value, and boolean expressions are not possible at the left hand side of an assignment. Further study would be required, and I will be very reluctant to do so, as the current regime stands for clarity. Any unnecessary ruling is a source for errors, and writing the assignment digraph instead of a simple equals sign is not at all a real burden.

### Void references and fault items in expressions

As the void reference refers to nothing, it is something like the empty set and has any and no property at the same time.

If it were forbidden as an operand for any operator in an expression and could only be assigned to a variable in order to void its reference, this would have been a sensible decision.

But for pragmatic reasons, the void reference as operand for an arithmetic operator always is treated as a zero for additions and subtractions, and as an empty string for a string operator. Even if both operands are the void reference, the result is a number (for addition and subtraction) or a string (for catenation).

Arrays and rows are dynamic and provide a void reference for non-existent or erased elements, which eases programming a lot, in particular as dealing with buffer boundaries is known to be error prone.

If then the void value were not allowed in arithmetic operations, the use of array elements as counters would require tedious programming for counting occurrencesof strings:

```
x =: arry{str}
? x = ()
    x =: 1
|
    x =+ 1
```

instead of

```
arry{str} =+ 1
```

Moreover, if an element (row or array) is requested for a void variable, a void reference is provided. The programmer still can check first if the bunch exists before to attempt to obtain an element???? HOW?

The scheme chosen is simple and ensures that the result of an operator always provides the expected kind of item (or a fault item). Note that other extensions, like treating the number 1 as true, or a non-empty string as 1 in a multiplication, was rejected in order to keep the rules simple.

As TUF-PL does not have exceptions, but instead uses fault items, a function within an expression may return a fault item instead of an item of the expected kind, in particular with operators with two operands and the case that two fault items are the operands. If only one or the only one is a fault item, it is simple just supply that fault item as result. If both are fault items, there is no sensible rule which one to pass, and what to do with the other one, as fault items must be acknowledged before being discarded.

Thus each function return is tested for a fault item, and if it is, skips all further expression evaluation and immediately returns the fault item as the function result.

### Conditional and logical expressions

In a logical expression, the following boolean operators may be used:

```
&   for logical and
|   for logical or
~   for logical negation
=   for logical equivalence
~=  for logical antivalence (exclusive or)
```

The implication must be expressed using:

```
~x | y  for the implication x→y
```

The terms in logical operators can be either comparisons, or variables and functions providing an item of the boolean kind (or void, see below).

As with all other expressions, all operands, including function calls that might have side effects, are evaluated, before the logical operator determines the result.

The C programming language defined a special property of the logical and (&&) — which since has often been considered advantageous — that makes the logical *and* an exceptional case. For any other operator, both sides must be determined before the result can be obtained. Only for the logical *and* the result is already known to be false if one operand delivers a false value, and C specifies that in this case the second operand is not evaluated at all..

This allows to write in C:

```
    if (x != 0 && x.field > 0)
        ....
instead of
    if (x != 0 ? x.field > 0 : 0)
```

or, better readable

```
if (x != 0)
    if (x.field > 0) {
        ....
    }
```

to clearly indicate a conditional evaluation.

The major disadvantage is the inconsistency, in particular that the operator is no longer commutative, and the only advantage is more compact code on the expense of creating a special case. It may have its merits in C, because it is often necessary to check if a pointer is null, before taking a field value, as otherwise the programme would crash. This is not necessary in TUF-PL, where the field of a void is just void, and void counts as zero (and false) in arithmetic and comparisons. Other programming languages have it intentionally undefined, witch is no remedy, as still the logical *and* needs special attention.

Thus, in TUF-PL, in all expressions all operands are evaluated; if this in no desired, nested conditions must be used (or for some common cases the guarded comparisons, which clearly flag the points of difference.).

Complex cases should be broken down into smaller pieces anyhow; efficiency is not gained by complex expressions, only unreliability.

Also, the use of iterators often makes the structure clear, as it separates the loop conditions from the loop actions.

Finally, a conditional expression is very often used to avoid using a void reference, e.g. to obtain a field, an element of an array, etc. Originally, all those operations were not allowed for a void reference, and lead to a fatal runtime error, so that they must be guarded. However, this has been changed to a more liberal regime: Nearly every (read) operation is possible with a void reference, and returns a void reference. E.g. getting an element of a row for a variable that contains a void reference will give a void reference, which is the same as would be given if an index for a non-existing element of an existing row would be given.

To avoid using the integer numbers `0` and `1` or the strings `true` and `false` — even if memory space is the same — for clarity, a *boolean* item can be used, of which only two exist, namely *true* and *false*, denoted as `?+` and `?-`.

As mentioned, void is considered false, if in a boolean context.

A variable or function result may used in a logical expression, but only if returns a boolean item.

This leads to the following pattern for a function returning a boolean result:

```
#Letters = 'abcdefghikjklmnopqrstuvwxyz'
string (inp) is a word:
    n =: count in string inp from 1 many of #Letters
    ? n = inp.count
        :> ?+
    :> ?-
some function:
    ? string 'word' is a word
        ... yes
```

In order to avoid the necessity to assign unset attributes a false value, the void reference is accepted as a false logical value. No other value is a boolean false, in particular not a number zero, an empty string, etc.

Note that the boolean false item is not the void reference, thus a function can return a three-valued logic of *true*, *false* and *void*, the latter something like *unknown*:

```
res =: some function
? res = ()      \ check for void first, as void is treated as false
    ... not applicable here, etc
? res           \ = ?+
    ... yes, positive answer
|
```

```
    ... no, definite negative answer
```

Although the boolean items can be assigned to a variable, the current compiler does not allow to assign or return a boolean expression; it is simply not implemented yet.

The unary boolean operator has very high precedence, in order to bind tightly to variables, thus, the negation of a comparison must be written with parenthesis:

```
? ~(i < 5)
? ~ check something     \ not needed for a single function call
```

There are no operators for the other possible boolean two-valued function, e.g. the implication, which must be expressed as `~a | b`, etc.

As it is very often the case to check a function result for a fault item, the `.isfault` attribute returns a boolean; true if it is a fault item, false else, so that the pattern can be:

```
res =: function may return a faulty item
? is res faulty
    ... error handling
    confirm res fault
```

Because there are only two boolean values, these are statically allocated. If you want to have longer names, you can use named literals:

```
#True = ?+      \ or even #T = ?+
#False = ?-
```

Avoid global variables, as these must be cleared before termination.

### String expressions

The primary String operators are:

```
=   equal
~=  not equal
_   catenation
__  catenation with a blank between, i.e. '_ ' ' _'
=_  string append
```

The choice of the full stop for catenation would not only be conflicting with the attribute and field notation, it also does not help reading a line with mixed variables and string constants.

Some languages allow *interpolation* of strings, such that variables are marked inside and automatically replaced once the string is used. Such a mechanism is often use in localisation of error messages:

```
File &filename invalid
Ungültige Datei &filename
```

As the ampersand (&) is the only character with a special function inside a string, and everything that is not a valid (and known) HTML entity (ending in a semicolon) is left in the string unchanged, it can nicely be used as a parameter indicator.

Using the the replace function of the standard library provides a solution that is flexible, fairly usable and does not need extensions:

```
msg =: "File &filename. invalid"
msg =: replace each '&filename.' by fn in string msg
```

Note that it is not useful to just replace `&filename.` with a preprocessor by `"_ filename _"`, as this would split the message text. Thus, a string replacement is necessary anyhow,.

The programmer is free to choose the parameter syntax, e.g.:

```
msg =: "Error copying from «source» to «target»"
msg =: replace each '«source»' by srcfn in msg
msg =: replace each '«target»' by tgtfn in msg
```

While it is very comfortable to simply de-reference the variable names, the disadvantage is that the same message at different places may require different variable names, and parameter names understandable by the translator are often not the variables names used.

Using array literal syntax and the replace function with arrays is not really more readable:

```
msg =: "Error copying from &source. to &target."
msg =: replace multiple { '&source.': srcfn, '&target.': tgtfn } in msg
```

While this is more compact, the later use of message catalogues is not simple, as the equals sign and anything that follows until the point have to be removed to create the list of messages. Also, the compiler must use the expression scanner now inside a string.

Extending TUF-PL with the hat (^) as a replacement operator in conjunction with relaxed array literal notation would allow to write:

```
msg =: "Error copying from &source. to &target." ^
    { source: srcfn, target: tgtfn }
```

However, this is not likely to be implemented unless the language becomes very popular.

The string catenation can also handle numbers, and converts them using a standard format without leading or trailing blanks. Thus, one can write:

```
p =+ 1
print 'p=' _ p
```

For debugging, often the following patterns are used:

```
debug print 'p="' _ p _ '" '
debug print 'ary{' _ ai _ '}="' _ ary{ai} _'" '
```

It would be rather helpful to write instead

```
debug print ?p
debug print ?ai __ ?ary{ai} __ ?row[ri]
```

or

```
debug print <p>
debug print <ai> __ <ary{ai}> __ <row[ri]>
```

Instead of using a Macroprocessor, most compilers will accept this shortcut directly. Then, `?x` is — inside an expression — a string operator, that produces the code

```
'x=' _ x
```

### String comparison

Strings are designed as rows of UTF code point numbers, thus string comparison concerns code points.

For equality, as the UTF-8 encoding according to the standard is unique, a common byte-by-byte comparison is done and works as expected: If two strings are equal, all corresponding code points are equal, and vice versa. This is also true if raw strings are compared to non-raw strings; they are always not equal (unless the string is marked falsely raw). As this comparison is presumed to happen seldom, there is no check for rawness yielding false whenever raw is compared to not raw.

Thanks to the excellent UTF-8 coding scheme, order comparisons, e.g. greater or less, can be done on a byte-by-byte basis, yielding a result as if the code points were first expanded to integers and then compared. This is the way the order comparison (< etc) works in TUF-PL, comparing code points, and thus is independent of the locale environment. However, order comparisons of strings marked as *raw* with UTF-8 encoded strings are highly suspicious and often cause unintended results. Thus, such an order comparison is a runtime error. Order comparisons of both raw strings are not an error and done bytewise, and also if one or both strings are known to be pure ASCII. Note that the rawness of strings is marked at creation and never changes because a string is immutable. However, the `.AsRaw` attribute of a string provides a copy of the string that has the `.IsRaw` attribute set, and just a copy of the reference if the string is already marked `raw`. Pure ASCII strings are also copied.

If comparing strings according to a locale is desired, a library function must be used:

```
compare string (str1) to (str2) under locale (locale):
\ returns -1 for less, 0 for equal and +1 for greater
compare string (str1) to (str2) under locale (locale) case ignored:
\ same as before, but letter case ignored, i.e. A=a
```

If the locale is void, the current locale is used.

If strings are compared several times to each other, the locale controlled comparison each time transforms the source strings such that the transformed strings, when compared bytewise, deliver the desired result; if e.g. two strings should be compared ignoring case, all small letters are transformed to upper case, and the results are compared. This intermediate string can be obtained by the function

```
transform string (str) under locale (locale):
\ returns a raw string that can be compared directly
```

The resulting string is marked `raw`, because its contents is undefined for any other purpose than comparing such strings. (It seems not necessary to provide a special flag just for this purpose.) If the locale is void, the current locale is used, and the resulting string may be different in further calls.

While it would be nice to have the current locale stored automatically as an attribute for a string, this might be useful in very few cases only, so the overhead is not justified. In these cases, a bunch may be used to associate the locale to a string.

There may be a function to add an attribute to a copy of a string; this requires a copy of the string, as strings are immutable, including their attributes:

```
copy (str) adding attribute (attr) value (val):
\ create a copy of a string, and add a user attribute
```

Strings cannot be compared to numbers, rows, or the void reference; any attempt to do so gives a run time error.

As with arithmetic, the shorthand `s1 =_ s2` is the same as `s1 = s1 _ s2`. Some time in far future the compiler and run time may use preallocated strings (*string buffers*) to speed up the process.

To prefix a string, explicit assignment must and can be used:

```
x =: p _ x
```

As strings are immutable, a copy is created anyhow and finally assigned, so there is no overlap problem.

No operators (only functions) are provided to access parts of a string, except the single character at a specific location, using row notation, which returns a string with a single character, not the integer equivalent of the character (nor Unicode code point). This notation cannot be used as a target of an assignment, thus to change a single character of a string, normally it is combined from the head before, the string with the new character, and the tail after it. A library routine

```
replace in string (str) character (pos) by string (repl):
```

will do so more efficiently, in particular if the strings are large.

**Bit operations**

As TUF has the concept of arbitrary precision integers, bit operations on negative integers are conceptually difficult, as the number of bits to be used is not fixed.

This is not a problem with the `and` and `or`, as long as the numbers are not negative, using the remainder of a division by 2 as bit extractor in a straightforward but inefficient way (these are candidates to be implemented in the host language, unless big integers should be supported:

```
binary (x) and (y) :
    ! x >= 0 & y >= 0
    z =: 0
    ?* x > 0 & y > 0
```

```
        z =: z * 2
        bx =: x %% 2
        by =: y %% 2
        ? bx > 0 & by > 0
            z =+ 1
        x =: x // 2
        y =: y // 2
    :> z
binary (x) or (y) :
    ! x >= 0 & y >= 0
    z =: 0
    ?* x > 0 | y > 0
        z =: z * 2
        bx =: x %% 2
        by =: y %% 2
        ? bx > 0 | by > 0
            z =+ 1
        x =: x // 2
        y =: y // 2
    :> z
```

If there is a library function to extract a bit (which is programmed here just for clarity), another solution — still for non-negative numbers — might be:

```
\ test if the n-th bit (bit 0 is the first) is set
is bit (n) set in (w):
    ! n >= 0& w >=0
    w =: w // 2^n
    :> w %% 2
\ and
binary (x) and (y):
    ! x >= 0 & y >= 0
    z =: 0
    n =: 1
    i =: 0
    ?* x >= n & y >= n
        k =: is bit i set in x + is bit i set in y
        ? k = 2
            z =+ 1
        n =: n * 2
        k =: k * 2
    :> z
\ or
binary (x) or (y):
    ! x >= 0 & y >= 0
    z =: 0
    n =: 1
    i =: 0
    ?* x >= n | y >= n
        k =: is bit i set in x + is bit i set in y
        ? k < 2
            z =+ 1
        n =: n * 2
        k =: k * 2
    :> z
\ xor
binary (x) xor (y):
    ! x >= 0 & y >= 0
    z =: 0
    n =: 1
    i =: 0
    ?* x >= n | y >= n
        k =: is bit i set in x + is bit i set in y
        ? k = 1
```

```
            z =+ 1
        n =: n * 2
        k =: k * 2
    :> z
```

Because in the commonly used two's-complement representation a negative number is the bit inversion plus one, the bit inversion is the negative minus one:

```
binary (x) inverted:
    :> -x - 1
binary (x) negated:
    :> 1 + binary x inverted
```

This extends to the exclusive or:

```
binary (x) xor (y):
    a =: binary x and y
    b =: binary x or y
    :> binary a and (binary b inverted)
```

Some bit operations can be conveniently expressed by arithmetic operations, in particular shifting by division or multiplication of two and its powers[7].

For the classical boolean bit operations there are library functions (to be implemented):

```
binary (x) and (y)
binary (x) or (y)
binary (x) xor (y)
binary (x) invert
```

Two additional functions allow to count bits:

```
count one bits in (x)
binary log (x)      \ = ld x, e.g. ld 3 = 2, ld 4 = 3, ld 0 = 0
```

To check if a non-negative number is odd, division or bit functions may be used:

```
is (x) odd using remainder:
    ? 1 = x // 2
        :> ?+
    :> ?-
is (x) odd using binary:
    ? 1 = binary x and 1
        :> ?+
    :> ?-
```

To access the bits as a row of bits, there are two library functions for non-negative integer numbers:

```
row of bits from integer (x)
integer from row of bits (y)
```

The first one creates a row of integer values `0` or `1` from the integer number, indexed starting from 0, the highest index corresponding to the highest bit set:

```
x =: 257
y =: row of bits from x
! y.first = 0 & y.last = 7 & y[0] = 1
z =: integer from row of bits y
! z = 257
```

Thus, `(row of bits from integer i).count` returns the number of bits in i, and `(row of bits from integer i).last` the highest bit number (zero based).

**Operator precedence**

The operator precedence just gives the order of evaluation in an expression, but does not ensure that only combinations are found that do not generate fatal errors. In particular, the various kinds are

mostly one-way: the string catenation will convert numbers to string, but the numeric operators do not process strings. Comparisons and boolean operators generate boolean values, but only the catenation operator, again, will convert them back to strings.

Some binary operators of equal precedence bind from left to right, i.e. `a+b+c = (a+b)+c` and `a-b-c = (a-b)-c`; these are (not allowed for all others):

```
| & + - *
```

Because the division slash is not in the list, neither `a/b/c` nor `a/b*c` nor `a*b/c` are allowed, although the latter expression would be independent of order.

Note that the symbols for the statement types, i.e. guard (`?`) etc, as well as assignment symbols like `=:`, `=+` are not operators.

Operators (except comparison) in ascending precedence (b=binary, u=unary):

```
1   b  |
2   b  &
3   b  < > <= >= = ~=
4   b  _
5   b  + -
6   b  * / // %% /% etc
7   u  - ^ ~
```

No operators are literal denoting symbols, i.e. quote, accent and tic, as these have closing correspondents. Also the digraphs `=:`, `::` etc, and the function template terminators are not operators. Note that `$#` is a special variable with context-dependent references and not an operator.

Array and row indexing as well as field and attribute notation (the dot) is not an operator; it is part of a variable denotation.

The (round) parenthesis for grouping and priority change are not regarded as operators.

**Bunch comparisons**

For bunch comparisons, no operator is provided, as the comparison is too much dependent on the application. However, equality comparison with the void reference is allowed as with any other item. Comparison with any other reference result in an run time error; while equal references imply equal contents, the opposite is not true, so the comparison has been banned. If the programmer is unsure, the kind can be tested before.

Some attributes, i.e. the number of elements, could be compared directly:

```
? r1.Count = r2.Count
```

A function to compare two rows might read as follows:

```
compare bunch (r) to (s):   \ true if same number and same kind
                            \ run time error if the elements
                            \ with the same index have different kind
  ? r.kind ~= s.kind        \ we are not comparing rows and arrays
    :> ?-
  ? r.count ~= s.count      \ same number of elements?
    :> ?-                   \ no
  ?# i =: r.scan            \ give all indices of r
    ? r[i] ~= s[i]          \ fatal error if comparison not defined
        :> ?-               \ unequal
  :> ?+                     \ ok, all indexed fields are the same
```

## 2.9. Lists and list assignments

This feature is not yet implemented, but has high priority

A list is a sequence of list elements, separated by commas. As the comma is a separator, two (or more) in a row are treated as one, and leading or trailing commas do not create empty list elements, which

thus do not exist.

A list can be used:

- instead of an expression as a list of expressions: `move to 3,5`
- on the left hand side of an assignment: `a, b =: 1, 2`
- in a function template (`move to (x,y):`)

If used instead of an expression, the list elements are aggregated into a row (see Bunches (aggregates)), and each element assigned the reference provided by the corresponding expression:

```
a =: 1, 'x', sin 5
! a.count = 5
! a[1] = 1
! a[2] = 'x'
! a[3] = sin 5
```

If a list of expressions is used as a parameter in a function call, the row is passed as parameter:

```
func call using x, y
\ is equivalent to
temp =: [2]
temp[1] =: x
temp[2] =: y
func call using temp
```

So to provide a function with a variable number of parameters, use a single argument and a list when calling it.

Because different kinds of arguments are resolved inside the language; the `print` function may accept strings and rows, in particular lists:

```
print (arg):
    ? is arg a row
        ?# i =: arg.scan
            print arg[i] nonl
            print ' ' nonl
    ? arg .= 0
        print string from integer arg
    \ whatever to do to print a string
```

Compilers may support a list as a parameter:

```
funct (a, b, c):
    ....
```

which is a shortcut for (see list assignments below):

```
funct (li):
    a, b, c =: li
```

In this case, an advanced compiler might bypass the creation of a row and create extra parameters, which would be equivalent to

```
funct (a) (b) (c)
```

### List assignments

If used on the left hand side of an assignment, the list elements must be valid left hand side destinations in assignments, and the right hand side must be a row item or just a list. It is expanded to individual assignments of the elements:

```
a, b, c, d.fld =: row
\ generated:
a =; row[1]
b =: row[2]
c =: row[3]
```

```
d.fld =: row[4]
```

Lists can be used at both sides of an assignment:

```
a, b, c =: 1, 'x', sin 5
! a = 1
! b = 'x'
! c = sin 5
```

When a row is assigned to a list of variables, void references are not suppressed, to keep the mechanism simple (e.g with lists as parameters, see above):

```
a =: []
a[1] =: 1
a[3] =: 3
x, y, z =: a
! y = ()
\ the same in one line:
x, y =: 1, (), 3
! y = ()
x, y =: 1, , 3
! y = 3
```

Note that a pair of commas does not create a void element; it must be explicitly written.

The array is started with the smallest index:

```
a =: []
a[-1] =: -1
a[0] =: 0
a[1] =: 1
! a.first = -1
! a.last = 1
x, y, z =: a
! x = -1
! z = 1
```

In order to avoid unexpected data losses, the number of row elements created so far (i.e. .last-.first+1) must not exceed the number of variables on the left hand side. (This as well as the retaining of void members may be changed in future versions, as there is not yet any experience in use.)

If void entries shall be suppressed or part of a row used, library functions can be used:

```
shrink array (ary):         \ copy without void elements
clip array (ary) from (start) upto (end):
```

Permutations may be written as

```
a, b, c =: b, c, a
```

If the compiler does not create a row item, all right hand list elements are first assigned to individual temporary variables, and then these to the left hand variables. So there is a difference to the compact notation:

```
a, b =: b, a        \ exchange
a =: b; b =: a      \ propagate old b to both
```

# 3. Items

Items are integer numbers, rational numbers and floating point numbers, booleans, character strings, rows or arrays, chunks or functions.

## 3.1. Fields and Attributes

Because attributes of items are an important concept, the details follow first. For a first understanding,

this section is not necessary; you may skip to Exact (integer and rational) Numbers.

Attributes are characteristic values of items, and each item has at least the kind attribute:

```
i =: 1
print i.kind        \ prints "integer"
```

An attribute of the item that a variable refers to is obtained by placing a point immediately behind a variable and then a word naming the attribute. Some attributes can also be obtained via functions, but often attribute notation is better readable, e.g. for constraints.

Attributes exist for immutable as well as mutable items.

As attribute are intrinsic values, there is no way to set them directly; so they are invalid on the left hand side of an assignment. As attributes are deterministic, one can create an item with a given attribute. Naturally, equal items have equal attributes (and thus there is no attribute providing the memory address).

Fields allow to store extra user information in mutable items (rows, arrays, bunches, fault items). A field of an item is selected by a dot after a variable name, followed by a tag word as a field name; thus the syntax is the same as for attributes. (It is often useful — if not necessary — to use a helper variable, as the notation might be restricted to variables; this has so far only made programmes clearer.) So the field name is determined at compile time, and fields can be used on the right hand side of an assignment. For storing information indexed by strings determined at runtime, use Arrays.... Any item reference can be saved and retrieved using field notation; in particular, organisational information can be attached to bunches this way.

Retrieving a not yet set field is void, and any field can be removed by setting it void; after that, it cannot be distinguished from a never set field (and the item it did refer to is freed, if this was the last reference). The void reference gives void for all fields (the only one that returns void for the `.tag` attribute) and attempts to set a field if the target is void or otherwise immutable, is a fatal runtime error. Otherwise, a field be set by any item reference (unless they are reserved fields).

Logically, equal items have equal fields; the definition of equality ensures this. Of course, the may exist items that are not equal, but where all attributes are the same.

Due to their nature, names of attributes are defined in the documentation (localisation included), while names of fields are selected by the programmer.

Some special items have reserved fields; these can be changed in an assignment (or by a function) and obtained using field notation. This concerns special items, as byte chunks, those used by file and other stream I/O, and fault items; thus they will generally be called *system items*. This is not different from libraries, where creation of an object normally returns a bunch (row or array) with context information stored in predefined fields, allowing the user to save extra information in fields with other names[8].

Originally, all field and attribute names were case independent, and attribute names had priority over field names. This may be harmful, e.g. for maintenance, thus the rules are now:

- Field names are case dependent and must start with a lower case letter.
- Attribute names are case independent an should start with a capital letter; using a (leading) small letter delivers the attribute value as long a no field with this name has been set.

So once only field names that begin with a small letter and attribute names with a capital letter are used, both are truly separated. Not only for historical reasons, this text will still use lower case for attributes.

Note that the attribute `.scan` is a scan function (see Index scans) and may return a different value each time used. As the functions `scan (), give indices of ()` do the same, this attribute will be dropped sooner or later for incoherence with the attribute concept.

Some attributes are common to all kinds of items; they never provide void (except for void):

| name | function | description |
|---|---|---|
| `.bytes` | `bytes used by ()` | size of storage occupied (bytes; 0 for void) |
| `.kind` | `kind of ()` | kind of item as string (empty string for void) |
| `.usecount` | `usecount of ()` | usecount (0 for void, >1 for all others) |

| | | |
|---|---|---|
| `.isfault` | `is () fault` | true only for a fault item, false else |

The attributes for numbers are explained in detail in their section; as they are regarded not mathematical functions, their functions are not prefixed with `math`:

| name | function | description |
|---|---|---|
| `.abs` | `abs ()` | absolute value |
| `.den` | `denominator ()` | denominator (1 for integers, > 1 for rationals) |
| `.frac` | `fraction ()` | fractional part `abs (x - ##(x.int))` |
| `.int` | `integer ()` | integer part (towards zero, signed) |
| `.num` | `numerator ()` | numerator (signed) |
| `.rsd` | `residue ()` | residue (numerator of fractional part) |
| `.sign` | `sign ()` | sign, i.e. +1, -1 or 0 |
| `.float` | `float ()` | closest floating point number or `#!` if out of range. |
| `.isbig` | `bignum ()` | true if arbitrary-precision number exceeding wordsize |

The attribute `.isbig` returns true if it is an arbitrary-precision number exceeding the wordsize, false for smaller accurate numbers, and void for all other, including floating point numbers.

For floats, `.den`, `.num` and `.rsd` are void, and `.float` is just a copy of the reference, as is `.int` for integer. The sign `.sign` is integer, as it can be easily converted to float again using the ## operator.

The attributes for strings are:

| name | function | Description |
|---|---|---|
| `.count` | `count ()` | number of characters (code points, not bytes) |
| `.scan` | `scan ()` | same as `from 1 upto s.count` |
| `.israw` | `is string () raw` | see section on strings |
| `.isascii` | `is string () ascii` | see section on strings |
| `.islocalized` | `is string () localized` | see section on strings |

The `.count` attribute for a string gives the number of characters (code points) in the string, so to convert a string to an array of single character strings might read:

```
row of characters from string (s):
    r =: [s.count]                  \ final size is known
    ?# i =: from 1 to s.count
        r[i] =: s[i]                \ or: code point of string s at i
    :> r
```

Mutable items (rows, arrays, chunks, fault items etc) have a common attribute:

| name | function | description |
|---|---|---|
| `.revisions` | none | count of changes since creation |

The `.revisions` attribute counts the number of modifications; thus it can be used to ensure consistency. It is used in scans to ensure the bunch is unchanged within a loop. For this purpose, only equality needs to be checked. When heavily used, e.g. in compute intensive algorithms to find a path, overflow may occur even if 32-bit unsigned integers are used. Comparing the magnitude is therefore dangerous.

Arrays have the following attributes:

| name | function | description |
|---|---|---|
| `.count` | `count ()` | number of allocated (non-void) elements |
| `.scan` | `scan ()` | gives indices, see below |

The `.count` attribute gives the number of non-void elements, i.e. just the number of indices a scan provides.

Rows have additionally to arrays:

| name | function | description |
|---|---|---|
| `.first` | `first index of ()` | index of first row element |
| `.last` | `last index of ()` | index of last row element |

See details for row expansion; note that for rows, `.count < .last - .first + 1`, as there may be allocated void cells skipped in a scan, which cannot happen for arrays.

There are two more special attributes always attached to mutable items (void for any other):

| name | description |
|---|---|
| `.tag` | identify (mutable) item |
| `.lock` | lock count; inhibits changes |

The `.tag` attribute is used to identify (classes of) items, set once by the function `equip (item) with (tag)` or within the braces when creating a row or array; a fatal runtime error will occur if not void before. Setting a tag is currently only possible for rows and arrays, other mutable items may return a not void tag. (The field notation to set a tag might still be found in sourcecode and accepted by the runtime system.) While any kind of item is possible as tag, strings are preferred. Tags are a major means for constraints and bunch functions,

Because of its central role, fields like `.tag`, `.tAg` etc are not possible.

The lock count is a means to protect a mutable item against changes. The `.lock` attribute is initially zero for mutable items, and void for others. If it is not zero, the item cannot be changed, i.e. neither values replaced nor new elements or fields added, with the one exception that the lock count can always be decreased and increased.:

```
lock item ():-          \ increase the lock count and thus locks the item
unlock item ():-        \ decrease the lock count and thus unlock the item.
```

where decrement returns a fault item if the lock was zero and is unchanged.

The lock count can be used as a semaphore for coordinating access.

Some constants of the currently running runtime system can be accessed as special attributes of the all-global variable `$SYSTEM`, see below on System variables and parameters

## 3.2. Exact (integer and rational) Numbers

Exact Numbers are arbitrary precision rational numbers, i.e. having a integer numerator and a positive integer denominator, where integer numbers are those with the denominator $1^{9}$.

Using assertions, the compiler could be given hints to optimize the code, and a runaway of numbers could be prevented.

To check the kind of item, in addition to the comparison operators and the functions, an attribute is defined for each kind, that returns true if and only if the kind of the item is:

```
.isvoid     void
.isint      integer (excluding rationals)
.israt      rational or integer
.isfloat    floating point (inaccurate) number
.isstring   string item
.isrow      is a row bunch
.isarray    is an array bunch
.isfault    is a fault item
.ischunk    chunk item
.isfunc     is a function reference
```

### Integral numbers

Integral numbers are arbitrary precision numbers, normally not limited to the wordsize of the system

that runs the programme.

Depending on the availability of a multi precision library, some runtime systems may have only 32 or 64 bit integer numbers[10]. If the size is sufficient, single (or double) machine words are used for arithmetic; except some rare environments, 64 bits should be available. The system automatically uses a arbitrary precision library; nothing has and can be done by the programmer. (Including to use machine words again if the number falls significantly below the limit.) The conversion library routines will also automatically return multi-precision numbers.

Professional programming should use assertions for the ranges of numbers, so that the compiler can detect at compile time that the range of numbers is not sufficient.

Addition, subtraction and multiplication of integral numbers have integral numbers as a result, thus there are no special considerations for these operators.

The division with a single slash (`/`) however, only yields an integer number if the divisor is a factor of the dividend; otherwise it is a rational number. If rational numbers are not supported, a fatal error will result. For this reason, the double slash (`//`) is provided as integer division, see below.

While the results are consistent among all proposals if the arguments are both not negative, the desirable extension to negative numbers is not as trivial as might be thought and has lead to different solutions.

As a generic symbol, `÷` is used here for the integer division, `†` used for the remainder; moreover, `n` is used for the dividend, `d` for the divisor, `q` for the quotient `n÷d` and `r` for the remainder `n†d`.

Raymond Boute [11] gives a detailed comparison on this subject and demands that the following two conditions hold[12]:

```
n = (n÷d)·d + n†d
|n†d| < |d|
```

He calls the first one the *division rule*; the second one could be called the *remainder rule*, which is given for completeness only, as no proposal violates it.

Within TUF-PL, it is also desirable that the integer operators are consistent with the rational numbers, which will be called the *rationals rule*:

```
n÷d = (n/d).int
n†d = (n/d).frac * d
```

A criterion that is not found in the literature is another consistency with rational numbers, which will however conflict with the *division rule*. For elementary fractions, we have the invariants:

```
n / d = (-n) / (-d)
(-n) / d = d / (-n)
```

So it would be expected that this holds for the integer division and the integer remainder too, which we will call the *fractions rules* for the division and remainder:

```
n ÷ d = (-n) ÷ (-d)
(-n) ÷ d = n ÷ (-d)
n † d = (-n) † (-d)
(-n) † d = n † (-d)
```

Boute considers three operator pairs, where the other one is defined using the *division rule*:

- truncation: the result is rounded towards zero
- flooring: the result is rounded towards the next lower integer, as proposed by D.E. Knuth.
- euclidean: the remainder is never negative

Using `//` and `%/` for the truncation pair, `/_` and `%_` for the flooring pair, and `/*` and `%*` for the euclidean pair, an example is sufficient to demonstrate the problems:

```
        //      %/      /*      %*      /_      %_
 29/6    4       5       4       5       4       5
-29/6   -4      -5      -5       1      -5       1
```

```
 29/-6  -4       5      -4       5      -5      -1
-29/-6   4      -5       5       1       4      -5
```

By definition, all satisfy the *division rule*, but only the truncation and flooring method fulfil the *fraction rule* for the division, and none follows the *fraction rule* for the remainder.

Nor surprisingly, the truncation division is the same as obtained from the *rationals rule*.

It would be possible to fulfil the *fractions rule* for the remainder, but at the expense of violating the *division rule*, with two alternatives:

```
n † d = (n%%d).abs
n † d =  (n/d).frac * d.abs
```

so that the remainder is either never negative, or has the sign of the quotient.

While the remainder that belongs to the truncation division has the sign of the dividend, there are also systems in which the remainder has the sign of the divisor, which violates both, the *division rule* and the *fractions rule* for the remainder and thus is not further considered.

Boute shows that for mathematical consistency, the euclidean division and remainder is the best choice, and Knuth's the second best choice, while the truncation division, although the most dominant one, is the least desirable selection.

However, he did not consider a very practical aspect: Which solution is the easiest to remember for the programmer, and might produce the least number of faulty programmes?

One of the — here easy to avoid — pitfalls is the to test a number to be even:

```
is even (x) WRONG:
    ? x %/ 2 = 1
        :> ?-
    :> ?+
is even (x):
    ? x %/ 2 = 0
        :> ?+
    :> ?-
```

With this in mind, the mathematical desirable is the least intuitive one, as it violates both *fractions rule*, even if a modified *rationals rule* might be obtained.

Both, the euclidean and the flooring division have the major disadvantage that not only the sign, but also the absolute value of the remainder depends on the signs of the divided and the divisor. Knuth's flooring division additionally delivers four different results for each sign combination, which is hard to recall during programming.

Another practical criterion observes that negative divisors are rather seldom, for which consultation of the language manual might be justified, but that for positive divisors the result should be fairly easy to predict. Here, again the truncation division wins, because the rule that the sign of the result (for all divisors) is that of the dividend could be kept in mind fairly easy.

This finally rules against the mathematical desirable versions, sticks to the commonly used truncated division, and leaves as alternatives:

- Either follow the *division rule* and remember that the remainder has the sign of the dividend, violating the *rationals rule*.
- Or follow the *rationals rule*, and, for simplicity, just use the absolute values of the operands, and violate the *division rule*.

However, to provide more than two operators is not really an effort, so TUF-PL follows the example of the few programming languages that leave the choice to programmer (and thereby keeps him alerted to check the manual), where the first three pairs follow the *division rule*:

- `//` is the truncated division and `%/` the corresponding remainder
- `/*` is the euclidean division and `%*` the corresponding remainder, which is never negative
- `/_` is the flooring division and `%_` the corresponding remainder
- `%%` is the magnitude remainder using the absolute values of the operands

As might be apparent, the choice of `/*` reminds that the definition for the mathematical modulus only uses the multiplication:

```
n = q * d + r
```

And the underscore hints to flooring, etc.

### Rational numbers

Using literals (constants) for integer numbers, rational number literals (constants) are build from integer literals, by evaluating constant expressions at compile time. Thus, an approximation for *e* is *27183/10000*. Note that the literal `3.1415` denotes an floating point number, although it is de facto the rational number `31415/10000 = 6283/2000`.

Using attributes, the numerator and denominator as well as other values can be obtained:

- `.num` for the numerator (signed integer)
- `.den` for the denominator (positive integer)
- `.int` for the integral part (signed integer)
- `.rsd` for the residue, see below (signed integer)
- `.frac` for fractional part (signed rational, .abs < 1)
- `.sgn` for the sign, `+1`, `-1` or `0` (signed integer)
- `.abs` for the absolute value or magnitude (rational, >= 0)
- `.float` for the floating point equivalent of an exact number

The corresponding functions are:

- numerator (rat)
- denominator (rat)
- integer (rat, float)
- residue (float)
- fraction (rat)
- sign (num)
- abs (num)
- float (num)

Thus, we have

```
x = x.num / x.den = x.int + x.frac = x.sign * x.abs
x.den > 0
x.sign = (x.num).sign = (x.int).sign = (x.frac).sign
```

The integral part is rounded towards zero, i.e. truncated division as normally provided by the hardware, which conforms to every day use, in that `-17/7 = -2 3/7`, where the fractional part of a number binds stronger than the unary minus, i.e. `- 2 3/7 = -(2 3/7) = -(2 + 3/7) = -2 -3/7`.

The numerator of the fraction in the mixed representation seems to have no generally accepted name, so we use *residue* for it:

```
x.rsd = (x.frac).num
x = x.int + x.rsd/x.den
```

However, the residue of a division of two integer numbers is not always the remainder or modulus of the division, see below.

Calculating with rational numbers benefits from reducing to lowest terms (*kürzen* in German) where the greatest common divisor (GCD) of numerator and denominator is calculated, and both are divided by it. This is done before the rational number is stored, because in TUFPL, numbers are immutable, and there are no clear performance benefits if the reduction would be postponed. So there is always

```
gdc(x.rsd, x.den) = 1
```

If the result of an arithmetic operation yields a rational number with a denominator of 1, an integer number is provided instead. As integer numbers provide 0 for `.rsd` and 1 for `.den`, these can be used like rationals if desired.

Note that due to the immediate reduction to the smallest denominator, currency values that require at

most two places in the fraction, may have `.den = 0`, `.den = 2`, .den=5', etc, a denominator less or equal to 100 and divisible by 2 or 5. E.g `54/10` is `27/5`, so the denominator is `5`.

Neither a scaling factor nor floating point numbers with decimal exponents are yet envisaged as part of the language. So it remains to use integers denoting cents and use constraints to forbide the use of rationals.

Some examples to illustrate the attributes:

```
        .num    .den    .int    .rsd    .frac   .sgn
 21/6    7       2       3       1       1/2    +1
-21/6   -7       2      -3      -1      -1/2    -1
 21/-6  -7       2      -3      -1      -1/2    -1
-21/-6   7       2       3       1       1/2    +1
```

Some TUF-PL compilers and runtime systems may not support rational numbers at all, or only as 32-Bit numbers, so that the programmer has to fall back to the usual dyad of integer and floating point numbers. See Comments and Pragmas for information how to ensure at compile time that rationals are available.

The demand for rational numbers seems to be rather low; possibly, because we are trained in problem solving using integral and floating point numbers only.

Strictly spoken, rational and integer number items are never equal — because each rational is always converted to an integer if the denominator is 1, and thus an arithmetic operation will never provide a *whole* rational. But as rationals can be intermixed with integers in arithmetic (as opposed to floats), comparison of accurate numbers will never surprise.

## 3.3. Approximate (Floating Point) Numbers

For many technical and statistical problems, approximate numbers are sufficient, as provided by floating point numbers, that are the practical equivalent of *real numbers*. Mathematically, they are rational numbers with rounding errors, so they are *approximated* numbers, in contrast to the *exact* integer and rational numbers. As of convenience, the term *floating point number* or *float* is used.

Literals for floating point numbers are written using the decimal point, i.e. `3.1415`, not the decimal separator from the locale. The forms `.5` or `7.` are not valid and must be written as `0.5` or `7.0`. According to the IEE 754 standard, floating point number arithmetic could produce positive and negative infinite, and two special NaN (*not a number*) values. These are denoted by the digraphs `#+` for +∞, `#-` for -∞, `#~` for a quiet NaN and `#!` for a signalling NaN.

A rational or integer number is not converted automatically to a floating point number; this could be done as follows:

- by the `.float` attribute
- by the ##-operator
- by the library function `float (x)`

All methods are equivalent and return:

- a copy if it is a float,
- the nearest float if it is an integer or rational number,
- 0.0 if void
- void otherwise

Although void is treated as zero in additions, the addition of two voids is always an integer, so the following would not work unless the explicit conversion to float would return 0.0 for void:

```
x =: ()
y =: ()
z =: x + y
! z = 0             \ result is integer 0
z =: ##x + ##y
! z = 0.0           \ result is float, as ## returns 0.0 for void
```

While the multiplication of void with any number is a runtime error, it can be circumvented by using

the conversion, as a float is allowed.

The library function might not work and should not be used in constraints and assertions.

Note that integer literals are treated the same. Unless the constraint system is properly used, the compiler has no information about the contents of a variable, and cannot issue a warning:

```
x =: 3
y =: 10.5
print x * y             \ runtime error, as x is integer
print x.float * y       \ ok
print ##x * y           \ ok
print (float x) * y     \ ok
print (float 3) * y     \ ok
print y * float 3       \ ok
print float 3 * y       \ runtime error, equivalent to float (3 * y)
```

Note that it is `(float x)`, because it is a function, and neither `float(x)` nor `(float) x`, and that function arguments are scanned greedily.

If the argument is already a floating point number, all three variants do nothing than return the item unchanged.

Some aspects function of floating point numbers are provided as attributes:

- The `.frac` attribute returns the fractional part, still a floating point number, with the sign of the argument.
- The `.int` attribute returns the integer part, i.e. largest integer number which is less or equal the floating point number, using absolute values, as an integer number, with the same sign as the argument.
- The `.round` attribute returns the integer with the smallest difference to the floating point number.

While the assertion `x = x.int + x.frac` theoretically should always hold, no such confirmation could be found specifications of the mechanisms used.

To get the smallest integer larger than the floating point number, use:

```
i =: r.int
? r > i.float
    i =+ 1
```

Rounding per `.round` is equivalent to:

```
round (fn):
    rv =: fn + 0.5
    :> rv.int
```

If the runtime does not provide unbounded integers, a fault item is returned if the integral part is too large. If this is undesirable, use the `+BigNum` feature flag to assert that the integer is large enough in any case. Otherwise, the program specification should specify the maximum expected value, and the floating point number must be checked against this value before. Adding an assertion will have the compiler checked the situation:

```
i =: f.int
! i.abs < 10^20         \ compile time error if no integers of this size
```

Loop to calculate the square root:

```
x =: a
y =: x * 1.01       \ thus y > x
?* y > x            \ monotonically falling until sqrt found
    y =: x
    x =: (x + a/x) / 2.0
print 'sqrt(' _ a _ ')=' _ y
```

(Note that $(x+a/x)/2 = x-(x^2-a)/2x$, so it's monotonically falling while $x^2>a$.)

Obviously, for an integer number, `.frac` is always zero.

For floating point numbers, the attributes `.den`, `.num` and `.rsd` are void. While for `.abs` and `.frac` floating point numbers are supplied, the attributes `.int` and `.sgn` return exact (integer) numbers, because it is easy to convert them to floating point, as in

```
! x = ##x.int + x.frac
! x = integer x + fraction x
! x = ##x.sgn * x.abs
! x = ##(sign x) * abs x
```

## 3.4. Strings

Strings are immutable sequences of characters. The number of characters in a string is practically unlimited, and the number of different characters (code points) is given by the Unicode standard, which is more than 1 million. As strings are character oriented, the `.Count` attribute returns the number of characters (not the number of bytes).

String comparison for equality is possible using the equals operator (=). Lexical comparison operators for *greater* etc. can also be used; in this case, the numerical value of the Unicode code points are used. If one is a substring of the other one, the longer one is greater. No locale is used by operators; thus surprises like `"A" = "a"` cannot occur. To compare using a locale, use standard library functions (which are still to be defined).

String literals are enclosed in either double quotes (") or single quotes ('), the former transformed by message localization. There is no way to denote the encoding in TUF-PL sources; it is assumed that the encoding of the source code file is known to the compiler, and that the complier converts to UTF-8. Using ASCII characters only together with HTML entities is strongly recommended, as it is independent of the encoding.

Strings can be joined using the binary catenation operator _ and the append assignment =_, which could be seen as a shortcut for `a =: a _ b`. The append assignment should be preferred whenever possible, not only for readability of the program, but also as a chance for an optimizing compiler to use a string buffer.

Unlike other operators, the catenation operator accepts numbers as operands and converts them to strings using a standard format. Other representations require use of a standard library function like `format (num) using (format)`. void items are just ignored; items of other kind produce a fatal error.

As strings are output anyhow, performance of string allocation and copy for long strings is often not relevant. Otherwise, collect the partial stings in a row and use a standard library function, that may be written in the host language and will preallocate the result string:

```
join strings of row (row):
    res =: ''
    ?# i =: row.scan
        res =_ row[i]
    :> res
```

Using this library function, there is no need for string buffers.

If argument lists are available, a function `join` could compose strings:

```
str =: join "&sqrt;(" , 2.0, " = ", (sqrt 2.0)
```

In contrast to the underscore string catenation, such a function could do more specific actions.

Each character in a string can be accessed using row index syntax, starting at 1, and returning void if out of range or zero as index; the result is a one-character-string to allow comparison to a string literal:

```
s =: 'abcdef'
! s[3] = 'c'
t =: 'cdefghi'
x =: s[4]
! x = t[2], x .= ''
! s[0] = ()
```

Note that this notation may only be used as a source, not as a target of an assignment, as strings are immutable.

Negative indices are allowed and count from the end, thus `s[-1]` is the last character, the same as `s[s.count]`, and it is always `s.first=1` (if the string contains at least one character).

Short strings (at leasts upto 5 bytes) are normally handled as efficient as integers, thus there is no need to have items of different kind for strings and characters.

To obtain the integer equivalent of a character, called *code point* in Unicode, library functions are are provided, that either return the characters individually or create a row out of each character:

```
character code point of (string) at (position):
    returns the code point as integer,
    or void (not 0) if position is not in string

row of character code points of (str):
    rv =: [str.count]
    ?# i =: from 1 to str.count
        rv[i] =: character code point of str at i
    :> rv
```

Unicode allows the zero code point, which is supported. This does not mean that strings are meant for processing binary data.

The opposite is done by

```
string from character code point (int):
    If the integer 'int' is a valid Unicode code point,
    i.e. between 0 and 1,114,111,
    a string of one character is returned.
    Otherwise, it is a fatal runtime error.

string from character code points in (row):
    rv =: ''
    ?# i =: row.scan
        rv =_ string from character code point row[i]
    :> rv
```

Converting a string to a row of single character strings, when really needed, is done by:

```
string row from string (s):
    r =: [s.count]          \ preallocate with s.count cells
    ?# i =: from 1 to s.count
        r[i] =: s[i]        \ assign string slice to row element
    :> r
```

To convert numbers to strings and vice versa, some routines are in the standard library:

```
integer from string (str):
    :> integer base 0 from string str
integer base (n) from string (str):
    \ if base n is zero, C language integer constant format is used
    \ returns fault item if string does not start with a number
float from string (str):
    \ converts leading decimal floating point number
number (n) as string:
    ? n .= 0
        :> format n using '%d'
    :> format n using '%g'
```

The library routine `format (n) using (f)` should be regarded as an intermediate solution, there is no checking of the format, so this must be used with great care:

```
format (n) using (f):
    \ uses a single printf-format to convert
```

To convert an integer to hexadecimal digits, use e.g.:

```
format i using '%2x'
```

### ASCII strings

Strings may have the '.IsAscii attribute, if all code points are below 127. As of the standard, this includes control characters below 32, in particular the tab character. Some library routines are more efficient if the string is indicated as ASCII.

However, absence of this attribute does not mean that it is definitely not ASCII, and for efficiency reasons, testing the `.isascii` attribute just gives the flag that is internally kept for the string. If a authoritative answer is required, the library function:

```
check if (string) is ascii
```

returns either the original string, or a copy, where the `.isascii` attribute is freshly checked.

There is no attribute to indicate what is sometimes called *printable*, i.e. is ASCII and has no other control characters than tab, newline and carriage return, but a library routine:

```
is string (str) printable:
    ? ~ str.ascii
        :> ()
    cnt =: count in string str many of '&tab;&nl;&cr;01234...'
    :> cnt ~= str.count
```

### Raw strings

Normally, strings are sequences of Unicode characters, internally encoded using UTF-8 conformant byte sequences.

But line oriented input from files, pipes or commands may obtain byte sequences that are not valid UTF-8 encoded strings. To allow the string handling functions to trust that encoding, the `.israw` attribute is set, if the string has failed the UTF-8 verification. Raw strings are byte sequences with code points 0 to 255 that can be passed as a reference and written out again. If the `.isascii` attribute is set, only the code points 0 to 127 are used in the string; but if this flag is not set, the string may still contain only ASCII code points, although most library routines check the result accordingly.

Some string processing operators and functions may reject to use raw strings, in particular in combination with Unicode strings, because the results will normally be useless. An exception is the catenation operator, that only marks the result as Unicode conformant, if both strings are such; otherwise, the byte sequences are catenated, and the resulting string is marked raw. This was considered useful, as this operator is often used in creating debug and error messages, that should work also under abnormal conditions, even if the results may be garbled. Note that the `i`-th character of string `s` via the notation `s[i]` delivers a one-character string (of the same rawness), not a code point number.

There are no plans to provide slices, use the `clip` functions instead. Note that strings are immutable, thus it is not possible to overwrite part of a string; this might have been a real inefficiency in former times, but nowadays the simplicity of immutable strings seems more valuable. Good real world examples where other languages like PYTHON and JAVA perform better are welcome.

As a substitute, the Chunks of bytes item may be used.

Attaching encoding information to a string is considered unnecessary complicated, as the string can be converted to Unicode once the encoding is known.

Two library functions allow recoding of strings:

```
recode string (rawstr) to Unicode using (coding):
    Recode the byte sequence of 'rawstr' to Unicode,
    using the encoding given by 'coding',
    and return a new string accordingly.
    If 'coding' is the voidref or 'UTF-8',
    the input is checked for UTF-8 conformance;
```

```
        in this case, if the input was not marked raw,
        a copy of the reference is returned.
        The input is always treated as a raw string,
        even if it is not marked as such.
        This allows any input string to be recoded.
        If an input byte is not a valid code point
        a fault item is returned.

    recode string (str) to raw using (coding):
        Returns a raw string derived from the string 'str',
        converting code points to byte values
        according to 'coding'.
        If 'coding' is the voidref or 'UTF-8',
        an exact copy marked raw is returned.
        If the source string is raw,
        or contains code points outside the table,
        a fault item is returned.
```

The coding parameter is normally a string, that selects a predefined coding; if it is unknown, a fault item is returned. The parameter may also be a row that is indexed by code points between 0 and 255 in both cases, the encoder to raw will lookup values and return indices for conversion.

Note also that the recode functions change the byte sequence, as opposed to a check that a string conforms to the UTF-8 rules, marking it as raw if not.

The encodings `ASCII`, `UTF-8`, `ISO-8859-15`, `ISO-8859-1`, `WINDOWS-1252`, and `DOS-437` are usually available. Known encodings can be queried:

```
    give all known string encodings:
        enumerates all available encodings giving string codes as above,
        always starts with 'ASCII', 'UTF-8' and 'ISO-8859-1'.
    check encoding (enc) for string (str):
        Test if string 'str', treated as raw string,
        is a valid encoding for the code 'enc'.
```

Converting from ISO-8859-1 to Unicode is particularly simple and always supported, as the code points are the same.

Unfortunately, ISO-8859-15 and ISO-8859-1 have the same code points, only different graphemes at 8 places; so they cannot be detected from inspecting the bytes. The WINDOWS-1252 encoding is a superset of ISO-8859-1 (not -15), but has only 5 undefined code points. DOS-437 has no unused code points.

Note that checking a valid WINDOWS and DOS encoded string for UTF-8 encoding often tells it is not UTF-8, as the latter requires certain sequences of byte patterns corresponding to seldom used special characters, that are not common in most texts.

The following function enumerates all possible encodings for a string:

```
    give encodings for string (str):
        ?# enc =: give all known strings encodings
            ? check encoding enc for string str
                :> enc
        :> ()
```

Another often used encoding is UTF-16, that uses pairs of bytes to encode most Unicode characters. It can often be detected by the byte-order mark that starts the file. It can be converted to UTF-8 using the above functions.

Using raw strings for byte processing is discouraged, use Chunks of bytes instead.

### Percent-encoded UTF strings

In particular query strings returned from HTML forms use a rather simple and short-sighted encoding, where improper characters are encoded using a percent sign (%), followed by exactly two characters that are hexadecimal digits. To deal with Unicode characters, there was a proposal to use %uxxxx or

similar for each Unicode code point; it was, however, rejected. Instead, a Unicode code point must be encoded in UTF-8, and each byte encoded individually using the percent method. This, however, allows to encode strings that are not UTF-8 encoded.

So to decode such strings in TUF-PL is nasty, e.g. as the function to create a (single character) string from a Unicode code point does not just map the codes from 128 to 255 to a single byte.

One way to solve this is using byte chunks, where we can put small integers in the range 0..255 as a single byte into a byte chunk.

Or, we provide a library function that converts integers from 0 to 255 in one-character raw strings. While string catenation just combines the byte strings, the result marked as raw, if one of the source strings is raw; thus, the result must finally be checked and the raw marker removed.

Due to the widespread use of this encoding, it seems best to provide a function implemented in the host language:

```
decode percent encoded string (str):
\ replace all occurences of %xx with single bytes,
\ and then check the string for rawness
```

As percent encoding might produce not-UTF-8 strings, these are marked raw if this is the case.

### String input and output

The line oriented file interface as well as pipes, including shell command execution, returns strings, which may or may not be encoded in UTF-8 (including 7-bit ASCII, which is a proper subset).

This is controlled by attributes and fields for the system item that is a file descriptor; the following text needs to rewritten, as the their attribute is still used even if fields are meant.

Instead of passing all received lines as raw strings, each line is checked if it conforms to the UTF-8 coding rules, and if not, marked as raw. This delivers all UTF-8 (incl. ASCII) encoded input as standard Unicode strings. However, it does not ensure that other encodings are always marked as raw. Fortunately and by design, the chance that other non-ASCII encodings are valid UTF-8 strings is relatively small, see http://en.wikipedia.org/wiki/UTF-8#Advantages. (In UTF-8, there are no single bytes with the high-bit set, while the special characters in other encodings are often followed by an ASCII character, thus strings in other encodings seldom pass the UTF-8 test.) Because the byte sequences are unchanged, just checked for UTF-8 conformance, there is no need to have an option that returns only raw strings.

If a different encoding is known prior to reading the input, the recode function may be used to convert the supplied string to a Unicode string. For this purpose, the resp. function always treats the input string as a raw string, even if it is not marked so.

Providing an attribute telling the code name and having the recode done automatically would be nice, but is assumed to be used seldom and thus is not implemented.

To obtain legible strings even if the input is neither pure ASCII nor UTF-8 encoded, the `.Autoconv` option is on by default. Aside from checking for UTF-16 BOM codes and recoding accordingly, any string that is not UTF-8 conformant, is converted to Unicode from ISO-8859-1. The most often encountered problem will be that if the input was ISO-8859-15 instead, the euro sign (codepoint 8264, graphem €) is shown as the currency sign (codepoint 164, graphem ¤). As the WINDOWS-1252 encoding is a superset of ISO-8859-1, the additional code points are also converted. If any of the remaining 5 code points are encountered while `.Autoconv` is active, a fault item is returned that refers to the offending line with the field `.line`. Note that with auto conversion on, the input bytes are not necessarily as retrieved from the system, but may be changed due to the recoding. Also, there is no indicator to show whether the input was originally UTF-8, or was recoded.

So the general rule is:

- Leave `.Autoconv` on. Provides no raw strings, but may have code conversion errors and return fault items, in particular if the input is random binary (or DOS-437)
- Switch off `.Autoconv`, expect mixed raw and Unicode strings, and take full control of any desired recoding.

As most often the trailing linefeed is not needed, and neither the carriage return, two additional

attributes are set by default:

- `.NoLF` removes a line feed character at the last position
- `.NoCR` removes a carriage return at the end, but only if `.NoLF` is set and has removed the trailing linefeed before.

If the stream is written to, each string written is considered a line and thus terminated by a line feed on Unix-like systems. This is indicated by the `.DoLF` attribute, which is on by default and can be disabled dynamically. There are systems that terminate a line by ac carriage return symbol before the line feed; on this kind of systems, the `.DoCR` is initially on, and can be switched off on demand.

Note that the attributes are a language feature, so they are valid only for the stream handle concerned, and other streams writing the same file may behave differently.

## 3.5. Bunches (aggregates)

Rows and arrays allow to aggregate items; the term *bunch* is used for any of both. For blocks of binary data use Chunks of bytes.

A bunch has:

- an arbitrary number of fields,
- either a row of arbitrary items, indexed by integer numbers,
- or an array of items addressed by item references, normally strings.

A field is addressed in the common dot notation that is also used for attributes, i.e. `bunch.field`. Rows addressed with the common index notation, i.e. `row[i]` Arrays use swindled braces instead: `array{s}`.

To create a row, a pair of square brackets (`[]`), and for an array, a pair of braces is used:

```
row =: []
array =: {}
```

Allocation of memory is done automatically as required; for a row, a small amount may be allocated initially. While any bunch can be used just to store fields, the array should be preferred.

Fields are like attributes, in that they are accessed by names provided as words literals in the programme text. Thus, field names cannot be generated during run time of a programme; for this purpose, array indexing has to be used. Note that throughout this document, relaxed notation is used where attributes are treated as system provided fields, see Attribute and field alternatives

Note that `r.x` and `r[x]` or `r{x}` are different: the first is the attribute with the name *x*, the second a row and the third an array indexed with the value of `x`.

Fields allow bundle global variables in a global bunch; this makes the release at exit easier. As bunches can be used as objects, a global variable with fields is similar to a singleton object.

### Rows

Rows are called arrays in other programming languages. A large number of items can be stored and quickly accessed by an integer. There is always a smallest and a largest index used so far, they can be obtained by using the attributes `.first` and `.last`. All cells between `.first` and `.last` are allocated; so the code:

```
row =: []
row[-1000] =: -1000
row[2000] =: 3000
! row.last - row.first + 1 = 3001
! row.count = 3001
```

creates 3001 cells, because the index `0` is included. Rows are dynamically expanded if indices outside the current range are used, and all cells in between are allocated. Freshly created cells are initialized with the void reference; and expansion only takes place if a non-void item is stored. In contrast to arrays, cells are never deleted, even if a void reference is stored. There is no means to distinguish a cell with its initial contents from one which was overwritten with a void reference.

To print the contents of all cells, including those with void values, use:

```
?# i =: from row.first upto row.last
    print row[i]
```

The row scan does not provide indices where the row value is void:

```
?# i =: row.scan
    ! row[i] ~= ()
```

Arrays are associative memories, that use a string to index the cells:

```
ary =: {}
ary{'a'} =: 1
ary{'a longer string'} =: 2
! 1 = ary{'a'}
! 2 = ary.count;
```

For arrays, only those cells are allocated that have non-void contents; writing a void erases the cell; there is no means to find out if a cell had existed before under a given index.

While strings are the most often used to index an array, any kind of item can be used that allows the equality operation (which excludes floating point numbers). In particular, sparse rows could use an integer index, and allocate only the used cells.

Indices are treated the same if they are of same kind and equal. Note that equality comparison for anything other than numbers and strings often compares the reference only, and may give unexpected results. In particular rounded number (floating point) have intentionally no comparison for equality and thus cannot be used as indices.

To enumerate the indices of an array, the special attribute `.scan` is provided; to store into a new row, use

```
rw =: []
?# s =: ary.scan
    rw[] =: ary{s}              \ rw[] is rw[rw.last+1]
! rw.count = ary.count
```

Using swivelled braces for arrays might look a bit baroque, but see Unified Rows and Arrays below for reasons not to unify rows and arrays.

Often, only attributes are needed, so the choice between an array and a row is fairly arbitrary. As an array is the more general construct, because any item reference can be used, the use of arrays is recommended in these cases[13].

Note that indexing with a void reference results in a run time error, and does not automatically create a bunch, as otherwise the item that holds the reference would have to be changed.

Row cells are not sparse, so the code sequence:

```
a =: []
a[1] =: 1
a[500] =: 500
! a.used = 2
! a.first = 1
! a.last = 500
```

creates, as indicated, 500 cells, of which 498 are void. If this is a concern, convert the integers to strings and use an array.

The automatic resizing for rows only increases the size; the row is not automatically shrinked. There may exist a library routine `shrink row (row)` to discard the unused memory at the borders; otherwise, the row must be copied.

Reading a row element outside the bound just returns void and does not automatically extend the row; this is done only if a (non-void) value is assigned. Whether storage is already allocated, can only be determined by comparing the index to the `.first` and `.last` elements.

If a row has to be expanded, and there is not enough memory available, a fatal error will occur (not a fault item returned, but see below).

While this will often be a programming error, e.g. allocating an endless chain of rows, or a runaway index, there are situations in which a large row is needed and the programmer can deal with the situation that it is not available. To pre-allocate a row with a given number of cells, a library routine is provided, `new row with (n) cells starting at (z)`. It returns either a reference to the allocated row, or a fault item (not void) if there is not enough space available:

```
r =: new row with 1000 cells starting at 1
? r.isfault
    :> r
! r.count = 1000
! r.first = 1
! r.last = 1000
```

Note that this row will still be expanded dynamically as long as there is memory available; to prevent it and issue a fatal runtime error if the row is to be expanded, the `.lock` attribute can be used, which however also generates a fatal runtime error, but maybe earlier.

To ensure a certain size and immediately try to allocate the memory, the library routine `ensure row (r) has (n) cells` could be used that expands the size to `n` cells (behind the largest index) and returns a fault item if this is not possible. To shrink an array so that `.first` and `.last` have non-void cells, used the library function `shrink row (r)`. Note that even if a row has been expanded to a certain number of cells, this does not mean that there is enough space to allocate the items whose references are to be stored in the row.

### Arrays

In contrast to rows, arrays are logically sparse, using linked lists, binary trees or hash tables. The number of estimated elements might be used only as an indicator to use hashed tables, but will often be ignored. Hash tables are very unlikely to be used internally, as the problem is to find an appropriate hash function. Binary trees have a penalty for small number of cells, whereas the linked lists proved to be quite efficient, in particular as the last found or saved element is moved to the top; this has decreased runtime in some array-base applications by the factor three, in particular if there is a common pattern like:

```
? i =: array.scan
    ? array{i} = somevalue
        array{i} =: othervalue
```

Keys or indices for Rows may be any item, although normally strings should be used. Expect the runtime to treat only items of the same kind as equal (weak equality) if a value for a key is replaced or for a new index. So, if strings and integers are mixed, note that `1` and `1` (the string and the integer) are **not** equal:

```
ary =: {}
ary['1'] =: 1
ary[1] =: 2
! ary.count = 2         \ two different indices
```

Values assigned to bunch elements may be any item reference; assigning the void reference to an element deletes that element, i.e. cannot be distinguished from a not yet allocated element. Users might it find error-prone and difficult that all kinds of items can be mixed; this can be restricted using assertions, in particular constraints.

Looping through a row (bunch indexed by integers) can be done using a standard loop:

```
?# i =: from r.first upto r.last
  print 'r[' _ i _ '] = ' _ r[i]
```

This will also return indices for which the values are void; to provide only indices for which the value is not-void, each row has a built-in enumerator:

```
?# i =: r.scan
  print 'r[' _ i _ '] = ' _ r[i]
```

The bunch enumerator may only be used if the bunch is not changed until the loop terminates; otherwise, a fatal error is generated. The bunch is not locked; the enumerator just compares the version number, so it is possible to leave the loop prematurely.

Note that the scan function is not informed of loop termination. Thus, providing a locking scan like

```
used integer indices of (r) locked:
    ? r.count = 0
        :> ()
    ? $# = ()
        r.hold =+1
    ii =: integer indices of r
    ? $# = ()
        r.hold =-1
    :> ii
```

will leave the bunch locked if the surrounding loop uses a break to terminate.

Better the programmer writes the locks outside the loop:

```
r.hold =+ 1
?# s =: r.scan
    print r{s}
r.hold =- 1
```

so that using a break may alert him on the problem. Instead of locking, the scans use a revision count to ensure the bunch is not changed during enumerations.

Elements of a bunch are deleted by assigning void to it. Once set to void, it cannot be distinguished from an element that never has been set. If there is a need to do so, a neutral element like the number zero, the empty string, a logical false, or an empty bunch may be used.

A bunch with zero elements can be created by using the empty row literal `[]` or by deleting all elements:

```
?# i =: r.scan              \ loop over all existing non-void elements
  r[i] =: ()                \ delete each element by assigning the void reference
! r.count = 0               \ assert the row is empty now
```

Note that there may exist several bunches with all elements deleted; they are not collected to a reference to a single empty bunch item, so one has to compare the number of used elements (attribute `.count`) to zero.

From the user point of view, any index can be used, even negative ones. Those cells that have not yet been given a value (reference to an item), will return the void reference, independent of being allocated or not[14].

Using the `.first` attribute, the numerically smallest index can be queried, and the largest index `.last` is the smallest index plus number of elements (attribute `.count`) minus one. The number of indices may be zero, in which case the smallest index is greater than the largest.

Note that `a[a.last+1]` refers to the next free cell, thus

```
add (val) to (array) at end:
    array[array.last+1] =: val
add (val) to (array) at front:
    array[array.first-1] =: val
```

As adding a row element to the next larger index is fairly often needed, an empty index, i.e.

```
a[] =: new value to be added
```

may be used instead of

```
a[a.last+1] =: new value to be added
```

Note that only `=:` is supported, as `a[a.last+1]` has the void value before, so that using `=+` would give a

fatal error. Note also, that for an empty row, the first index used is one, not `row.last+1`, which would be 0.

Sparse rows are not supported via integer indices, only via string indices. While the bunch is automatically expanded, it is not automatically shrunk; this must be done by a library routine call, possibly using the number of non-void cells provided by the `used` attribute.

Note that indexing the void pointer as well as using an item other than an integer or string as index is not supported and will result in a run time error.

As rows and arrays can contain references to other rows and arrays, cyclic references can be created. This spoils the automatic storage deallocation, as such a chain is self-consistent, and will cause memory leaks. Thus, the standard runtime system keeps track of the number of allocated bunches, and will give an error message at normal termination if there are bunches left over.

This is particularly nasty if doubly-linked lists are used:

```
a =: {}
b =: {}
c =: {}
a.next =: b         \ b.usecount = 2
b.prev =: a         \ a.usecount = 2
b.next =: c         \ c.usecount = 2
c.prev =: b         \ b.usecount = 3
a =: ()             \ a.usecount = 1
b =: ()             \ b.usecount = 2
c =: ()             \ c.usecount = 1
\ whole chain is not freed; keep each other alive
```

The programmer must erase all backward links manually.

A library function could recursively walk through a net of bunches, flag bunches visited internally, and erase all references to bunches already visited. Then the whole net will be released if the last reference is out of scope.

### Tagged bunches

The `.tag` attribute of a bunch can be set by the user freely to a reference to any item, but only once, i.e. only a void can be overwritten.

It allows a kind of type-controlled programming, in particular if used with constraints and assertions, and also makes object-oriented programming simpler.

A tag can thus be any item, but commonly stings are used, in particular in a hierarchy notation, e.g. `de_glaschick_tuf_lang_env`. The use of the delimiter is free, but the underscore (instead of a dot or blank) is recommended for Tagged bunches....

Its canonical use is for constraints (see Assertions and Constraints), like in

```
! env: @.tag = 'de_glaschick_tuf_lang_env'
do something on (env: env):
    ...
start all:
    nb =: {}
    nb.tag =: 'de_glaschick_tuf_lang_env'
    do something on nb
```

Besides the use in assertions and constraints, tags are a major means to associate functions with bunches, supporting a kind of object oriented programming, see Tagged bunches..., where the strings are more likely to be word lists e.g. `env glaschick`.

For the programmer's convenience, the tag name may be given during creation within the brackets:

```
na =: {'de_glaschick_tuf_lang_env'}
nr =: [de_glaschick_tuf_lang_env]
```

If the tag is just a word (thus the use of underscores), the string quotes may be omitted. As it makes no

sense to localize the tags, do not use double quotes. Most compilers will reject such an attempt. Note that numbers give the estimated size, and thus cannot be used as tags in this notation.

## 3.6. Chunks of bytes

To cope with arbitrary binary data, the item kind called a *byte chunk* is provided. As the name says, these are chunks of unstructured bytes in memory. In contrast to strings, there is no inherent interpretation, and byte chunks are mutable. Their primary purpose is to access binary files in the filesystem. Byte chunks are a variant of a bunch, i.e. support similar attributes and also allow arbitrary fields set and obtained.

Byte chunks are created, accessed and modified by standard library functions only, and references to byte chunks are used like any other item, including the memory management that keeps them until the last reference has gone.

Byte chunks are allocated in one piece and cannot grow or shrink. Standard functions allow to get or put strings, binary coded numbers, or byte chunks. As no standard serialisation is defined for arrays and rows, these cannot be get from or put into byte chunks directly.

As usual, the `.bytes` attribute returns the number of bytes used for data. It is always greater zero, because an empty chunk cannot be created.

Chunks have one reserved field (not attribute, as it can be changed):

| name | change | description |
|---|---|---|
| `.byteorder` | YES | set or get byteorder |

Freshly created chunks are not initialised, and to inhibit the use of stale data, a fill level is automatically set to the last byte changed. If data is put into an area not yet written to, any not yet written bytes are filled with zeroes. The fill level is available via an attribute; to keep the number of attributes low, the `.count` attribute name is used for this.

Function using byte chunks use the general patterns

```
... chunk (chunkref) at (offset) size (size) ...
... chunk (chunkref, offset, size) ...
```

the latter using a single parameter with a list; the choice depends on personal taste and may not be supported.

Thus access to byte chunks always uses an offset and a size, where the offset starts with zero for the first byte. The offset may be negative, in which case it is counted from the end, thus -1 is the last byte and 0 the first byte. Whether a size that exceeds the defined data is accepted, depends on the function.

A byte chunk can be created using a standard library function, which returns a reference to the byte chunk, or a fault item if the space was not available:

```
bcr =: new chunk of size (n)
? bcr.isfault           \ no memory
    print bcr.msg
    bcr.status =+ 1
    :> bcr
! bcr.bytes = n
! bcr.count = 0
```

The standard library functions to extract data from byte chunks are:

```
i =: get integer from chunk bcr at 32 size 4        \ signed
i =: get counter from chunk bcr at 36 size 1        \ unsigned
r =: get float from chunk bcr at 0 size 8           \ length 4 or 8
s =: get string from chunk bcr at 77 size 16
s =: get string from chunk bcr at 77 size 99 zero ends  \ null terminated
b =: get chunk from chunk bcr at start size len
```

These functions return references to new items with the data copied in the internal format of the corresponding item.

A byte order must be set for integers and floats (see `.byteorder` above). In order to reduce the chances for unnoticed data corruption, the size when getting a number must not exceed the initialised space (as returned by `.count`).

Floating point numbers are regarded as IEEE 754 single (32bit) or double (64bit) format, thus only the sizes 4 and 8 are guaranteed.

Integer numbers are regarded as binary signed in two's-complement notation. To extract unsigned numbers, the word *counter* is used, although the same effect can be achieved by adding 256 for a single byte, $256^2=65536$ for two bytes, etc, after extraction. Unless arbitrary precision numbers are implemented, the size must be between 1 and 8. For a single byte, setting a byte order is not necessary.

For string extraction, the first form copies the number of bytes given into a string, and then checks for UTF-8 encoding, setting `.raw` if that fails. The resulting string may contain zero bytes, in particular trailing zero bytes.

Because zero-terminated strings are often used in external data, the variant with the added term `zero ends` copies until a zero byte is found, which is not copied as TUF-PL does not need it. If there is no zero byte, copying stops when the end as indicated by the size is reached.

If a chunk is extracted from a chunk, a new byte chunk is created with the size given, and the bytes starting at the offset given are copied. If the size given exceeds the available data, only that data is copied, but the chunk allocated for the full size; the fill level is lower then.

To replace data in a byte chunk, the standard functions are:

```
l =: put integer (item) into chunk (tgt) at (offset) size (len)
l =: put counter (item) into chunk (tgt) at (offset) size (len)
l =: put float (item) into chunk (tgt) at (offset) size (len)
l =: put string (item) into chunk (tgt) at (offset) size (len)
l =: put chunk (src) at (soffset) size (slen) into chunk (tgt) at (toffset) size (tle
```

These functions return the offset of the next byte after the last byte accessed, i.e. `offset+len`, where `offset` and `len` might have been restricted (and negative offsets converted). If this number is equal to `.bytes`, the chunk is full now. As mentioned below, unused space below the area written is automatically zeroed.

A byte order must be set for integers and floats as above. In order to reduce the chances for unnoticed data corruption, the size when putting a number must not exceed the space available, else it is a fatal error.

Floating point numbers are binary floating point numbers in IEEE 754 single (32bit) or double (64bit) format, thus only the sizes 4 and 8 are guaranteed.

Integer numbers are formatted as signed binary numbers in two-complements; the value must in the appropriate range (e.g. -128..127 for a single byte), or a fatal error occurs. To write unsigned numbers, the form `put counter ...` is used for clarity, even if this could be done by just adding 256 etc. if the number is negative. For size 1, setting a byte order is not necessary.

Strings are not truncated to avoid accidental data loss, i.e. there must be sufficient space after the offset. To determine the space needed, the `.bytes` attribute of the strings should be used, not the `.count` attribute, as the latter counts characters. If the size given for the byte chunk is larger than needed, the remaining space is filled with zeroes, so a zero-terminated string may be created as follows:

```
str =: 'König'
put string str to chunk xx at 32 size (str.bytes + 1)
```

Note that TUF-PL strings may contain zero bytes, even if UTF-8 encoded, thus the above only guarantees a zero end byte, but not as the only one.

When copying between byte chunks, the target length must allow all source bytes to be copied, i.e. data is not automatically truncated. If the source is shorter than the target space, it is filled with zero bytes. The source size is the effective source size, taking into account the end of the byte chunk and the fill level. The chunks slices may overlap; in this case, it works like an extra buffer is used.

To create a new byte chunk by catenating the contents of two others, no library function is necessary nor significantly more efficient:

```
catenate chunk (first) and (second):
    nbc =: new chunk length (first.byes + second.bytes)
    pos =: put chunk first at 0 size first.bytes into chunk nbc at 0 size first.bytes
    put chunk second at 0 size second.bytes into chunk nbc at pos size second.bytes
```

An example to write a string with a leading string length in binary:

```
str1 =: "Hello, world"
bcr =: new chunk size 12345
pos =: zero chunk bcr size 1+str1.bytes
pos =: put counter str1.bytes into chunk bcr at pos size 4
pos =: put string str1 to chunk bcr at pos size str1.bytes
```

As byte chunks are bunches, attributes may be used to track positions and to allow shorter function templates. User attributes are used:

- `.getpos` to remember the next byte to get
- `.putpos` to remember the next byte to put
- `.getsize` to set the size of the next get
- `.putsize` to set the size of the next put

`.getpos` and `.putpos` are updated when used.

So there are functions with omitted parts following the general pattern:

```
... chunk (chunkref) at (offset) size (len) ...
... chunk (chunkref) size (len) ...
... chunk (chunkref) at (offset) ...
... chunk (chunkref) ...
```

An example is:

```
bcr.getpos =: 1036
get integer from chunk bcr size 4           \ signed
get counter from chunk bcr size 1           \ followed by unsigned
bcr.getpos =: 32
get float from chunk bcr size 8
bcr.getpos =: 77
bcr.getpos =: 1024
bcr.getsize =: 32
?# i =: 0 to 8
    cr =: get chunk from chunk bcr          \ adjacent slices
    ... process chunk cr
```

List notation makes no sense here.

### Byte order

When treating byte sequences in byte chunks as binary words, the byte order is relevant, i.e. whether the byte with the lowest number is the least significant byte (LSB) or most significant byte (MSB). The former is often called *Little endian byte order*, and the latter *Big endian byte order*, also called *network byte order* because most internet protocols use it. Each byte chunk has an attribute `.byteorder`, which can be `U` for undefined, `L` for LSB first, and `M` for MSB first, and is only propagated if a chunk is created from another one. It must be set before any integer or float is extracted or written; using the host byte order as default is not supported in order to avoid unintentional byte order dependencies. The `.byteorder` field can be set not only with `U`, `L` and `M`, but also with `H` for host byte order and `N` for network byte order. Reading back, however, always gives `U`, `L` or `M`, so this can be used to determine the host byte order.

Only the first character is used when set, and may also be a small letter, so

```
bcr.byteorder =: 'host'
bcr.byteorder =: 'LSB'
```

are both valid and work as supposed.

Using not a string with at least one character of the above set is a fatal error.

Note that `.byteorder` is a reserved field (lowercase), not an attribute, as it can be set by the user. However, a field must be returned unchanged;t Thus it is an attribute???

### Block IO

Byte chunks provide access to binary system files.

The functions to exchange byte chunk data with the file system are:

```
fd =: open chunked file reading (filename)
fd =: open chunked file overwriting (filename)
fd =: open chunked file appending (filename)
fd =: open chunked file updating (filename)
rcnt =: read chunked file (fd) to chunk (bc) at (pos) size (size)
wcnt =: write chunked file (fd) from chunk (bc) at (pos) size (size)
loc =: tell chunked file (fd) position
lco =: position chunkedfile (fd) start plus (loc)
rc =: close chunked file (fd)
```

The following example shows how to copy a file (with minimal error handling):

```
infd =: open chunked file reading "/dev/random"
outfd =: open chunked file writing ("/tmp/xx" _ process id)
bc =: new chunk of size 1024*1024
?*
    rcnt =: read chunked file infd to chunk bc at 0 size bc.bytes
    ? rcnt.isfault
        :> rcnt         \ not necessarily a fatal error...
    ? rcnt = ()
        ?>              \ break loop if end of file
    ? rcnt = 0
        ?^              \ repeat if nothing read, but not yet end of file
    wcnt =: write chunked file outfd from chunk bc at 0 size rcnt
    ? wcnt.isfault
        :> wcnt
    ? wcnt ~= rcnt
        :> new fault item with code 1 and message "Write failed."
close chunked file infd
close chunked file outfd
```

The `read chunked ...` function reads from the file as many bytes as are possible to the chunk determined by the position and size; here, the whole chunk is used. Note that while UNIX file IO returns 0 for end-of-file, and an error for a non-blocked IO that has nothing to deliver, the above function returns void instead of a fault item for end-of-file.

To transfer the data to another file, the `write chunked ...` transfers all bytes from the byte chunk slice, as indicated by position and size, but may fail to write all bytes and returns then number of bytes written.

Due to the similarity of byte chunks with UNIX files, a file can be memory mapped to a byte chunk. The library functions to do this are still to be specified.

## 3.7. Fault items

A programme may encounter two types of exceptional situations:

fatal errors:
: These are faults that cannot be repaired within the programme. Typically, these are logic flaws that were not detected during creation, or situations, where the situation is so defective that the fate of the programme can only be to die.

recoverable errors:

> These are exceptional situations that are considered not normal and require special measures. This is typically the case if an operation cannot provide the results normally expected, e.g. a read error during a file read. These are faults that lead to failure, unless special recovery measures are taken.

The traditional way to deal with recoverable errors in the C programming language is to indicate error conditions by return codes. However, it was observed that:

- ignorant (i.e. not knowing better) programmers don't care for the return codes
- otherwise, the returncode is often passed upstream, with high probability that in this chain the return code would be ignored, so that many error conditions were lost and lead to undefined behaviour

As an alternative, the exception model was introduced. A block has the option to handle the exception, or to pass it upstream. Now programmers could no longer accidental ignore error events and continue with code that runs under false assumptions. However, while the separation of normal program flow and error handling allows the former to follow more easily, an in-depth handling needs to bring together the lexically separate statements in the block of the normal program flow with that of the exception handling. Still, the major advantage is that error conditions are not lost and can be passed to the caller easily. The details of catching and unravelling exceptions is hidden from the programmer and requires a fairly complex runtime support.

In TUF-PL, exceptional conditions do not require new syntactical support; just a new kind of item, the fault item and a slightly different treatment for assignments are necessary. Eventually, it is not so different from exceptions, only integrated differently in the language. To avoid confusion with exceptions used in other programming languages, this term is not used in TUF-PL.

Fault handling is done by having a special kind of item, the fault item, which can be returned and processed like any other item. But the fault item has the special feature that a newly created one will not be silently lost if the value is to be freed, e.g. because the variable that holds a reference gets out of scope, or the function result is discarded. Instead, a fatal error will be the result, see details below.

As fault items are not intended to be used instead of bunches, there is no general collision probability of fields and attributes. Thus, all information is contained in fields (in lower case). These field names may not be localised:

```
.isfault    same as 'is (x) fault'
.checked    zero integer if not yet processed
.errno      error code number
.code       error code string, e.g. 'ENOENT', or void
.message    message string
.backtrace  row with backtrace information
.lineno     line number where the fault item was created
.prototype  prototype of the function that generated the fault item
.info       additional information
.when       to differentiate different sources
.data       for partially accumulated data before the error occurred
```

The programmer may add more fields and modify the above ones (except `.isfault`).

The function `is () fault` is recommended over the use of the field `.isfault`. Normally, fault processing is guarded by first calling it after a value was returned by another function.

The field `.checked` is set to integer 0 initially, indicating a not yet acknowledged fault item. To mark it processed, any other value is accepted; it can be incremented by a function or a field assignment:

```
fault (fault) is checked:
    fault.checked =+ 1
```

Done each time the error has been logged or otherwise acted upon, this allows upper levels to discard fault items with a large enough number. Thus fault items could be returned after marked as serviced to signal failures upstream, just for unravelling call chains *manually*.

While it is easy to write a function circumvent nagging messages as *unprocessed fault item*, its use is very limited and thus it is not in the standard library:

```
ignore fault (fault):
```

```
    ? fault.isfault
        fault.checked =+ 1
        :> ()
    :> fault
```

The `.backtrace` attribute, if not void, is a row representing the function backtrace (without parameters) at the point of error. Starting with 1, each odd element is a string that contains the function name, and each even element the corresponding source code line number.

As fault handling parts of programmes are normally guarded using the boolean attribute `.isfault` or the function `is () fault`, collisions are rather unlikely.

To create a new fault item, a library function is provided:

```
new fault item with code (code) and message (msg):
```

To deal with error code numbers, the standard way is to define named integer literals, e.g.:

```
#EPERM  = 1     \ Operation not permitted
#ENOENT = 2     \ No such file or directory
#EINTR  = 4     \ Interrupted
```

While possible, because the C library has to have these codes unique anyhow, the use of small strings, i.e. `EPERM` as error code would be preferred; if the code is just a string, or a second field with the code string (with the message text in addition) is used, is still to be determined.

A function that detects an uncommon situation returns a fault item instead of the normal result, which could easily be checked in the further programme flow to start error recovery.

As an example, take the following code fragment, printing the contents of a file:

```
print all from (fn):
    fd =: new stream reading file fn
    n =: 0
    ?*
       str =: line from file stream fd
       ? str = ()                       \ end of file ?
            ?>                          \ yes, terminate loop
       print n _ ': ' _ str
       n =+ 1
    :> n
```

If the file cannot be opened, the function `new stream reading file ()` returns a fault item, which the function `line from file stream ()` refuses to handle generating a fatal error.

If during reading, a read error occurs, `str` receives a fault item. The string catenation operator _ does not accept fault items, and causes the programme to stop with a fatal error.

To handle the fault items, the item kind could be checked; to increase readability slightly, a system attribute `.isfault` is defined for every item including the void item, which returns `?+`, the true value, only if the item is a fault item. Thus, instead of analysing function-dependend return codes, to catch the error, the `.isfault` attribute has to be checked:

```
print all from (fn):
    fd =: new stream reading file fn
    ? fd.isfault                      \ error?
        print "could not open " _ fn _ " reason:" _ fd.msg
        fd.checked =+ 1               \ mark processed
        :> fd                         \ tell upper level about the error
    n =: 0
    ?*
        str =: line from file stream fd
        ? str.isfault
            :> str                    \ pass read error upstream
        ? str = ()                    \ end of file ?
            ?>                        \ yes, terminate loop
        print str
```

```
        n =+ 1
    x =: drop file stream fd
    ? x.isfault
        :> x                    \ pass error upstream
    :> n
```

The less often encountered errors when reading and closing are simply passed upstream; only the standard open error is shown to be processed directly, and marked as processed.

Using a library function, the whole becomes much simpler:

```
print all from (fn):
    ?# str =: give lines from file fd
        ? str.isfault
            :> str                      \ pass read error upstream
        print str
        n =+ 1
    :> n
```

Of course, the calling function has more to do to sort out a fault item, unless it simply prints flt.message and flt.when.

When a fault item is created, it is marked as `not processed` by setting the `.checked` attribute to integer zero. In order to avoid accidental loss of unprocessed fault items, the attempt to discard such an unprocessed fault item (e.g. by assigning a new value to a variable holding the last reference) results in a fatal error. Once the status is not zero, the fault item can be freed automatically like any other item.

If the second to last line would not be present, and instead be written:

```
drop file stream fd
```

the void reference normally returned would be silently discarded. If something else is returned, discarding a non-void value will be a fatal error anyhow. It might be tempting to write:

```
x =: drop file stream fd
:> n
```

to get rid of any non-void return values, but it does not work for fault items, because no fault item with zero usecount is freed unless its status is not zero.

In order to help upstream error processing, any field can be set in a fault item; recommended is the use of `.when`:

```
print all from (fn):
    fd =: new stream reading file fn
    ? fd.isfault                    \ error?
        print "could not open " _ fn _ " reason:" _ fd.msg
        fd.checked =+ 1             \ mark processed
        fd.when =: 'open'
        :> fd                       \ tell upper level about the error
    n =: 0
    ?*
        str =: line from file stream fd
        ? str.isfault                   \ pass read error upstream
            str.when =: 'read'
            :> str
        ? str = ()                  \ end of file ?
            ?>                      \ yes, terminate loop
        print str
        n =+ 1
    x =: drop file stream fd
    ? x.isfault                         \ pass error upstream
        x.when =: 'drop'
        :> x
    :> n
```

The scan function to read lines of a file does just this, so the compact version would be:

```
print all from (fn):
    ?# str =: give lines from file fn
        ? str.isfault
            error print 'Failed on ' _ str.when _ ', reason: ' str.info
            str.checked =+ 1
            :> str
        print str
```

Note that the scan returns a fault item if the drop (close) fails, and does not just terminate the loop.

Another help for debugging fault items is the backtrace stored in the `.trace` attribute whenever a fault item is created, which is row of elements alternating between function name (string) and line number (integer).

The pattern:

```
x =: do some function
? x.isfault
    :> x
```

is quite often needed, but clutters the code with error processing, even if the condensed form is used:

```
? x.isfault; :> x
```

This can be avoided by using the error guarded function call `:!` to indicate that instead of void, a fault item may be returned, and in the latter case the function should return the fault item upstream unchanged:

```
:! do some function
```

In case of an assignment, the error bubble assignment `=!` allows to write:

```
result =! some other function   \ and return result if a fault item
work with result
```

instead of:

```
result =: some other function
? result.isfault
    :> result
work with result
```

It can clearly also be used in a scan loop, as it is just an assignment with a specific special effect.

Because not yet implemented, the digraphs `:?` and `=?` may be used instead of `:!` and `=!`.

Experience so far shows that there are not so many places where the fault item is returned without having added some information, and, in particular in iterators, the test is not so seldom delayed one statement.

In case that despite the arguments that follow someone insists on having the equivalent of try/catch blocks, the error guard `??` should be used:

```
print all from (fn):
  ??
    fd =: new stream reading file fn
    n =: 0
    ?*
        str =: line from file stream fd
        ? str = ()                      \ end of file ?
            ?>                          \ yes, terminate loop
        print str
        n =+ 1
    drop file stream fd
    :> n
```

```
    |
      error print 'Failed on ' _ str.when _ ', reason: ' ?@.info
      ?@.checked =+ 1
          :> ?^
```

Note that the same *else* notation is used as for a normal guard, as is must follow immediately to a guard and requires to pass context anyhow.

In this case, every assignment (and every function return value to be discarded) is checked for a fault item, and in case it is a fault item, execution continues in the error else block. Within the error else block, the special variable `?@` contains the reference to the fault item.

However, the above example is felt to be artificial; in most cases, the try-part of the block could be grouped in a function, and the example will become:

```
print all from (fn) block:
    fd =! new stream reading file fn
    n =: 0
    ?*
        str =! line from file stream fd
        ? str = ()                      \ end of file ?
            ?>                          \ yes, terminate loop
        print str
        n =+ 1
    :? drop file stream fd
    :> n
 print all from (fn):
    x =: print all from fn block
    ? x.isfault
        error print 'Failed on ' _ x.when _ ', reason: ' x.~info
        x.checked =+ 1
        :> x
```

A fault item is created via the library function

```
new fault item having message (msg) and code (code):
```

Note that the fault item mechanism is an affective countermeasure against a certain class of errors, cited from the close system call documentation:

> Not checking the return value of close() is a common but nevertheless serious programming error. ... (it) may lead to silent loss of data."

The same clearly holds for an error detected during reading the next line.

### Fault item special cases

Some peculiar situations may arise:

First, if a loop repeatedly returns fault items that are just collected in an array, the program will use up all memory, as with all other endless loops filling arrays, as the fault items are not discarded unprocessed yet. Runtime systems could limit the number of fault items created without marking a fault item processed in between, but this seems to be a fairly rare case.

What is the proper handling of fault items within an enumeration? Setting the enumeration variable to void and returning the error item will immediately result in a fatal error, as the iteration is stopped, but the return value discarded. So the fault item should be saved into the iterator variable and its previous contents in the `.data` attribute. If the caller does not process the fault item, the next iteration round will try to replace the old fault item with a new one, which will raise an unprocessed fault item error. If an iterator may return a fault item, it should assert that it is not called again with the error item in the iteration variable, as this would also prevent the above endless loop.

### Signals

UNIX signals cannot be mapped to exceptions, as there are none. As fault items replace exceptions, signals are honoured using fault items.

There are two kind of signals: synchronous, often called traps, like arithmetic overflow, illegal instructions etc; and asynchronous.

As traps occur if the program is buggy, they immediately terminate the program. Some people find it comfortable to use a try/catch bracket around calculations or other statement sequences that might crash, instead of carefully considering each source of error individually. From my point of view, the recovery of such error situations is either more complicated than detecting the error in the first place, or sloppy, leading to programs with erratic behaviour. Thus, catching traps is not supported in TUF-PL. In case an application must continue in case of errors of subfunctions, the latter must be put into processes of their own, as to protect the main process as well as other children processes from interference.

There are several options how to treat (asynchronous) signals, of which the last one is used:

First, all signals could be disabled. This may be fatal unless for short times only.

Second, signals could be mapped to a fatal error with stack trace, but no means for recovery. This rather handy when a program loops and must be stopped. It is, however, a bit unusual if a program requesting input is stopped by CTRL-C and then shows a stack trace. Note that using curses or readline, this behaviour will not arise.

Third, a signal could set a global flag that (and when) it occurred. However, this must be polled; but it is still not a bad solution if properly designed, i.e. ensured that the flag is inspected often enough. But a timer might still be needed because the programme is far to complex to ensure proper polling.

Forth, a signal could call a TUF function, which is restricted as described in the Unix manual. Unfortunately, the only thing this function could do, is to set some global variables or change an item that is passed with the intercept call. After that, it can just return and have processing resumed where it was interrupted, or stop the programme prematurely.

The pair `setjmp/longjmp` cannot be used directly, as it would result in memory leaks of all the items hold in the variables of the abandoned stack frames. As memory management uses reference counts, no garbage collection is available that would clean up memory use after the `longjmp`. A similar solution might come, in that each function returns immediately with releasing all local variables so far, until a fault handler is provided in a function.

A fifth option, which is standard for several signals, changes the return value of the next function return to a fault item composed by the signal handler. The original return value is put into the `.errdata` field, while the backtrace information was saved in the `.errbt`field. Thus, the evaluation of expressions and assignments continues as if no signal occurred. This is sensible, as the time (asynchronous) signals occur is not related to the program, so the time after such local processing is as good as the original time.

After the continued chain of expressions and assignments, either a return, a function call or a loop repeat may be next. A return just returns the fault item instead of the original value, which is saved in the `.errdata` field.

A loop repeat is treated like a return with a void value; i.e. the function returns immediately independent of the loop nesting. Function calls are also aborted and replaced by an immediate return of the pending fault item. When a function returns the fault item set by the signal handler, this information is cleared.

For this to work even with tight loops like:

```
?* 1 = 1
    \ do nothing
```

the loop code inspects at each round a global variable which contains either void or a pointer to the new fault item, and returns immediately this value if it is not void. The return code then returns this value if it is not void, thus working even if the function would return anyhow. If it is already beyond that code, i.e. returning anyhow, there are two possibilities:

- let the program run until the next loop or return
- check at each function call

The first option would not catch a run-away program that is just recursively calling functions *without using a loop*, i.e.

```
runaway1 (x):
    x =: 1
    runaway2 x+1
    print "never reached"
runaway2 (x):
    x =- 1
    runaway1 x
```

Note that

```
also runaway (x):
    ? (x // 2) = 0
        also runaway x+1
    |
        also runaway x-1
    print "never reached"
```

is not caught, because there is no loop, but

```
not runaway (x):
    ?* x > 0
        x = not runaway x
    never reached
```

will be caught, because it has a loop. However, the runaway programs will finally stop with a stack overflow anyhow; without using a loop or calling another function, this will be very quick. Unfortunately, some systems do not call the signal handler, as this would require some space on the stack, so there is nothing like a backtrace etc., unless a debugger is used.

Unless a stack overflow can be handled properly, there is no need to check during function call code for a pending fault item.

Unfortunately, this scheme is practically useless as it is, because any function can return a fault item, and either the program unexpectedly has a fault item where another one is expected, and traps into a fatal runtime error, or discards the value, which also traps in a runtime error for non-processed fault item.

So what needed is a scheme that allows a function (or block) to register for the reception of fault items; so it is similar to the try/catch mechanism.

It remains the question of how to cope with multiple signals, because signal races are a real source of unreliability. So, once the forced return of a fault item caused by a signal has been initiated, the next signal must not do the same until the programme signals the end of error handling by setting the `.checked` flag as usual. However, to ensure normal functioning during fault item processing, the code that does the forced return clears the location pointing at the fault item. The signal handler uses a second location having a reference to the current fault item caused by a signal, and checks if the fault item is still unprocessed. If it is processed, a new fault item can be injected; otherwise, it is not yet specified as if a fatal error is displayed, or a queue of unprocessed signals is kept and the next signal item dequeued if the current one is marked as processed.

Finally, there is the situation that library functions use system calls like `sleep` or `wait` or some blocking IO. In these cases, the system calls return after the signal handler returns, and sets errno, which will be recorded to a fault item; thus these are safe. However, some system calls like `fgets` do not return if interrupted by a signal during an attempt to read; it just repeats the read. Maybe using `setjmp/longjmp` should be used here; the current version honours the interrupt after some input.

It is admitted that it is not easy for the programmer to sort out fault items returned; and that it needs some discipline to process only the expected fault items and pass all others; as there is no language help in this as with some exception mechanisms.

### Fault items: conclusions

Thus, the net effect is nearly the same as with exceptions, but:

- has the mechanism described in the language itself
- allows flexible interception

- does not require wordy boilerplates at each level
- still prevents errors from going by undetected
- allows the programmer to easily add extra information

The disadvantage may be that the code is slower compared to the use of exception in C++, as the latter can unravel the call stack by some complex runtime routines only when needed, while fault items will do many checks for the normal program flow, and this sums up even if the checks are very quick compared to the other overhead.

Note that there are still conditions under which not a fault item is returned, because the error is considered non-recoverable, i.e. errors in the program logic than in the environment of the program. Clearly this includes the case that an unprocessed fault item is going to be destroyed. It also includes wrong item kinds, in particular providing voids instead of tangible data, the attempt to index rows with string, or to add strings to numbers.

# 4. Assertions and Constraints

Assertions and constraints are rather similar and start with an exclamation mark (`!`), basically followed by logical expressions. They allow to declare properties that are invariants during runtime of the program, and by this means may:

- help to stop faulty programs early, that do not behave as predicted
- allow an enhanced compiler to generate better code
- provide a discretionary type system

If all constraints and assertions are removed, the function of the program is unchanged; if they are present, additional runtime errors may occur and more efficient code generated.

As a rule of thumb, assertions are checked each time they are encountered and may generate runtime errors, while the effect of constraints depends on the facilities of the compiler, and generate only compile-time errors if the constraints are contradictory; depending on the compiler, they may be fully or partially ignored.

## Assertions

An assertion is a line in the body of a function that starts with an exclamation mark. In its basic form, the rest of the line contains a boolean expression that must be true, otherwise the program stops with a fatal runtime error:

```
! b %% 2 = 1            \ b must be an odd integer
! x.abs < 1'000'000     \ x must be a number less than ± one million
! x >=0 & x < 1'000'000 \ zero or greater, but less than a million
```

Any expression yielding a boolean result may be used. Function calls are seldom used, as the purpose is a quick sanity check for program invariants, e.g.:

```
! x.count = y.count     \ both must have the same number of elements
```

Note that two assertions following immediately are equivalent to using the AND operator & to combine both expressions:

```
! x >= 0
! x < 1'000'000
\ same as
! (x >= 0) & (x < 1'000'000)
```

A artificial example to use the OR | operator is:

```
! x = () | x > 0
```

Here, x can be void or be an integer greater zero; note that void can be used in any comparison and yields false (unless compared to another void).

Problems may occur in the — rather seldom — case that a variable may contain different kinds other than void, e.g. an integer or a string:

```
! x < 56 | x < '56'
```

This assertion is only passed if x is void; in any other case a fatal runtime error will occur, as one of the two is a comparison that is not allowed.

However, in most situations, the kind of variable is predictable, and the assertion should check this too.

If a function accepts different kinds as parameters (which is perfectly allowed), the processing must be split anyhow, as further processing is specific to each kind. Then, the assertions can be placed after the branch determination:

```
collect argument (x):
    ? is x string
        ! x < '56'
        ....
        :>
    ? is x integer
        ! x < 56
        ...
        >:
```

Advanced compilers can support a block depending on an assertion like in:

```
! 0 <= x & x < 1'000'000
    x =+ 5
    do some processing with x
    ..
```

The compiler might deduce from flow analysis that the first check on entry is `0<=x & x < 999995'`, and then there is no check necessary after the increment.

Also, the compiler could optimize the code and, because the numbers fit well into 32 bit integers, use machine instructions instead of item processing subroutines.

Assertions without a depending block are checked just at the place where they are written during runtime (unless optimised).

For better readability, a list of boolean expressions may be use instead of the heavy ampersand, and could be regarded as a short form for two immediately following assertions:

```
! x >= 0, x < 1'000'000
```

To allow more complex assertions that do not crash due to item mix, on could propose to get rid of the restrictive rules for comparisons and simply make them false, if the kinds differ etc. This is not appreciated, as it introduces more complex rules. A solution could be using the more expressive constraints of the next section.

## Constraints

Constraints are named declarative assertions and replace the type system used in other languages. While assertions are executable statements inside a function body, constraints bind a name to an assertion expression for a single variable. This name can be bound to a variable, which declares an implicit assertion each time the variable is changed. If the compiler can determine statically that the assertion is always true, no code will be generated.

The difference to a classic type system is not only that constraints are expressed for flexible, e.g. limiting the range of a number or the length of string, including a minimal length (which is evidently very seldom used, but shows the flexibility).

Also, the classic type system needs a mechanism to override the restrictions (normally called a cast) and even remove it completely. In this case, all checks are left to the programmer. In TUF, the checks are always generated, unless the compiler can determine statically that the constraints cannot be violated by the current assignment.

Constraints are syntactically very similar to assertions, i.e. the line begins with an exclamation mark, the constraint name (a word), a colon, and an assertion expression, where the item to be constrained is

represented by the scroll symbol @:

```
! n16:          \ define 16 bit non-negative integer number
    @ >= 0      \ implies integer
    @ < 2^16
! integer:      \ general integer number
    @.den = 1   \ integer numbers have a 1 denominator
    @.frac = 0  \ alternate indication
```

Constraint names have a separate namespace.

The given constraint for an integer is just an illustration; it is clearly better to write:

```
! integer:
    @.kind = 'integer'
```

which is one of the predefined constraints.

A variable can be bound to a constraint by prepending the name and a colon or an exclamation mark:

```
! int32:
    @.kind = 'integer'
    @.abs < 2^31-1
int32: y = x +1     \ the result must no exceed 32 bits
int32! y = x +1     \ same as above
```

Even if the code is cluttered with colons already, we prefer it to be less invasive than the exclamation mark, which admittedly would have been the logical choice.

The constraint binding may be used at any place, in particular in the parameter list of a function:

```
n16: advance integer (n16: x):
    x =+ 1
    n16: y =: x / 2
    :> y
```

A constraint binding is valid for the whole scope of the variable, independent of the position. To restrict it to the lexically following uses would create a false sense of protection.

Note that in the above example, the binding to y is useful to detect a problem early; as the function is bound to n16 in the template, the return will check the constraint towards the value For best readability, the first occurrence is used. Duplicate constraint declarations of the same name are allowed, but different names are not allowed (even if they are equivalent).

As constraints are declared outside function bodies, the expressions may not use local variables or parameters, and also no global variables; only literals (including named literals).

The constraint target is denoted by the scroll symbol (@), and can be omitted just after the colon, if nothing else could be meant.

To make constraint definitions more compact, a single line can be used, using a list separated by commas as in assertions, i.e. the following lines are equivalent to each other:

```
! n16:
    @ >= 0
    @ < 2^16
! n16: @ >= 0, @ < 2^16
! n16: @ >= 0 & @ < 2^16
```

Inside the expressions, already defined constraints may be used:

```
! i32: @.abs < 2^31         \ integer implied
! n16: i32:@, @ < 2^16
```

A tree can be thus declared like this:

```
! string: @.kind = 'string'
! tree:
```

```
        tree: .lhs
        tree: .rhs
        string: .op
```

So if variable is marked as a tree and has the field `.lhs`, this refers to an item that must obey the tree constraint; and if a field `.op` is present, it must be a reference to a string.

In the code fragment

```
tree: t =: {}
string: s =: 'xx'
t.op =: s
t.lhs =: t
t.op =: 0           \ creates a constraint violation
t.lhs =: s          \ the same
```

the last two lines could be flagged by the compiler as a constraint violation already at compile time.

If a variable is not declared as certain type, the compiler generates code to check the constraint at runtime, if necessary:

```
a function with parameter (x):
    tree: t =: {}
    t.op =: x
```

expands to (because the field `.op` is constrained to a string):

```
a function with parameter (x):
    tree: t =: {}
    ! x.kind = 'string'
    t.op =: x
```

If the parameter is constrained to a string as in:

```
a function with parameter (string: x):
    tree:t =: {}
    t.op =: x
```

the check will be done upon entry of the function, i.e.:

```
        a function with parameter (string: x):
        ! x.kind = 'string'
        tree:t =: {}
        t.op =: x
```

Note that constraint verification are applied only if a value is change, i.e. on the left hand side of an assignment.

Of course, if the function is private to the module, it can be checked statically that all invocations hold the constraint, and no check upon entry is necessary.

The notations:

```
! rowOfInts: integer: @[]
! arrayOfIntRows: rowOfInts: @{}
! arrayByStrings: @{string}
```

tell that the row has all integer values, the array only `rowOfInts` values, and that in `arrayByStrings`, only strings are used as indices. Logically, the integer indices of a row could be constrained too:

```
! i100: @ > 0 & @ < 101
! a100: @[i100]
```

Note that the body of the constraint has the assertion syntax, thus one may write:

```
inOrStr: integer: @ | string: @
```

Using tagged bunches allows to explain data structures using the *same tag* boolean operator `@=`:

```
! node: @.tag = 'node'      \ also: @ @= 'node'
    @.lhs @= 'node'
    @.rhs @= @
```

If a word is used as a constraint name, but not defined by a constraint declaration, it is treated as a tag constraint:

```
! aRow: @ =@ 'aRow'     \ can be omitted
    aRow: x =: [aRow]   \ compiler may allow [] instead
    aRow: y =: x
```

To allow more efficient analysis, the `.tag` attribute may be set only once, i.e. if it is not void, it is unchanged further on.

Just to restrict a bunch to a given tag name may be written as:

```
  @.tag = 'syz'
  @ @= 'syz'
  @'syz'
```

If the exclamation mark is in column 1, the constraint name is globally valid in the module, otherwise only in the block (and all dependent blocks) in which it occurs. Contrary to local assertions, they are valid independent of the location; putting them at the beginning is recommended.

The compiler might issue a warning that the assignment to y might violate the assertion for y in the following code:

```
! n16: @ >=0, @ < 2^16
! i7:  @.abs < 2^7
sample function (n16: x):
    y =: x / 2
    :> i7:y
```

Depending on the compiler, a compile time warning could be given if the user writes:

```
! int16: integer, @ > -2^16, @ < 2^16-1
advance integer (int16: x):
    :> x + 1
use it:
    advance integer 0.1
    advance integer -1
    advance integer 2^16  \ expect a warning
```

However, writing

```
advance integer 2^16-1
```

will often result in a fatal error message only, as the argument fits to the assertion, but the function body creates 2^16 and thus an overflow[15].

Note that constraints are much more expressive than standard declaration, as the diligent programmer can state exactly the ranges of the numeric variables; a banking application would declare:

```
#maxSavings = 1´000´000´000
! savings:
    @ >= 0                  -- 0 is also 0/100
    @ <= #maxSavings
    @.den = 100
```

telling that numbers with 12 decimal places for the integral part are needed and that they must be kept as rationals[16]. The compiler might use 64 bit integers and a fixed denominator of 100, or double floating point numbers and fix the fraction often enough.

If, however, the maximum above would read `10^12` or `1000´000´000´000´000` 64-bit integers would be required.

Note that during calculation, more digits might be needed. As assertion violations are not errors, but

programme failures, to sum up such values, either the standard built in arbitrary precision numbers may be used, and the result checked:

```
savings: add (savings: input) to (savings: balance):
    new = input + balance
    ? new > max Savings
        :> new fault item ....
    :> new
```

Or a new assertion must be used:

```
savings: add (savings: input) to (savings: balance):
    ! new < 2*maxSavings
        new = input + balance
        ? new > maxSavings
            :> new fault item ...
        :> new
```

As the compiler can determine that no overflow can occur with 64 bit integers, no overhead would be necessary.

This may seem to be unnecessary complicated, but allows to write programmes of high reliability, as the current situation tempts the programmers to ignore integer precisions. Even if the compiler is not smart enough and just uses arbitrary precision numbers, problems will be more easily found during testing.

Normally, declared assertions include the void value implicitly, as it is extensively used in the language. Global variables in particular are initialized with the void reference, thus it must be allowed. But a void reference is not the same as the number zero, most often a compiler will not optimize global variables that are numbers, although the values for NAN (not a number) for floating point numbers and the smallest number (i.e. 0x80...00) could be used for the void reference.

For strings, assertions could be used to limit the length of strings, when required:

```
! s7: @ .= "", @.count <= 100
```

To exclude the void reference, write:

```
! s7: @ ~= (), @ .= "", @.count <= 100
```

As the length of the string must always be kept additionally, a compiler might optimize `s7:` strings to occupy 8 bytes[17].

A constraint may include other constraints:

```
! nonneg: integer: @ & @ >= 0
```

To make reading and writing easier, the `: @` may be left out; just a constraint name is sufficient:

```
! nonneg: integer, @ >= 0
```

Declared assertions cannot be stacked, thus the following line is invalid:

```
a: b: x =: y        \ invalid
```

If `x` must obey `a` as well as `b`, a new constraint can be defined:

```
! ab: a, b
ab: x =: y
```

For the basic kinds of items, there are predefined constraints:

```
! void: @.kind = 'void'
! error: @.kind = 'error'
! boolean: @.kind = 'boolean'
! integer: @.kind = 'integer'
! rational: @.kind = 'rational'
! float: @.kind = 'float'
```

```
! string: @.kind = 'string'
! row: @.kind = 'row'
! array: @.kind = 'array'
```

The constraint name `any` is also supported, that denotes no constraint.

As the boolean expression may contain logical and as well as logical or operators, it is possible to describe a logical union of constrained elements.

If, e.g., an item can be an integer or a string, this can be expressed like this:

```
! IntOrString: @.kind = 'integer' | @.kind = 'string'
```

Using the above predefined constraint names, this becomes

```
! IntOrString: integer | string
```

To indicate that all elements of a row are integers is expressed as follows:

```
! RowOfInts:
    integer: @[]
```

Thus, the digraph `@[` followed by `]` denotes all elements of a row, and `@{ }` all elements of an array.

The digraph `.*` denotes all user defined fields that are not explicitly constrained, so we could require all fields to be constrained by:

```
! void: @.*
```

This effectively prohibits the free use of attributes, which may be very useful in some applications for high reliability. E.g., to limit a bunch to the fields `.first` and `.other`, use:

```
! limbun:
    any: @.first
    any: @.other
    void: @.*
```

The `.*` notation does not propagate, so we may have:

```
! limbunplus:
    limbun: @
    any: @.third
```

Note that a constraint for a specific element of a row or an array is not supported.

It is good practice to used tagged bunches in constraints, like in:

```
! exprNode:
    @.tag = 'exprNode'
    exprNode: @.next
    any: @.payload
```

To make the code easier to write and easier to browse, the digraph `@'` starts a string, that constraints the `.tag` attribute, as in:

```
! exprNode:
    @'exprNode'
    exprNode: @.next
    any: @.payload
```

Note that using localised strings as tags is considered very dangerous, as there is not yet enough experience with tags.

There is currently no field that returns the name of the constraint at run time, as constraints are conceptually treated at compile time only.

A language extension might drop the requirement for the scroll symbol `@` in constraints.

Constraints may seem onerous, if an operation temporarily violates the constraints, in particular during creation of items. If this is not caused by badly defined constraints, one solution is to assign the item to an unconstrained variable, do the necessary changes, and copy the reference back to a constrained variable. This is a direct consequence of the fact that constraints are connected to variables, not to items. Using factory functions is the standard measure for problems during creation. Another solution is to lock the bunch, as constraints are not applied to locked bunches.

There is currently no means to define a constraint that a function is a scan function, because:

- constraints refer to items, not to functions
- a scan function uses a special global variable to characterise

## Assertions for Bunches (rows and arrays)

Plain values like numbers and strings are not individualized by constraints, as they are immutable anyhow.

Bunches, however, can occur in nearly infinite variants, depending on the use of user attributes and the kind of references, if indexed. As long a bunches are processed by functions in the same module, the programmer may well keep track mentally the layout of the bunches used.

A library of functions that needs to keep context between invocations can either use small integers (handles) and a global variable with a reference to a bunch. Or it can create a bunch, return it to the caller, and expect to called again with such a bunch as argument. In this latter case, there is an inherent risk that the bunch is modified in a way not expected by the library.

One possibility is to set the `.hold` or `.lock` attributes, or both, to prevent accidental changes; however, the attributes are themselves not locked, so the bunches may still be changed.

maybe we need a mandatory constraint?

For bunches, a single assertion , assertions for bunches

It is necessary and sufficient to use the prefix at the lexically first occurrence, but it may be used at any place, provided the same one is used at each place.

Note that a function assigned to an element of a bunch can access the bunch itself, called `this` in other languages, by the symbol `@` (or, if enabled, `this`). Note the scroll is never part of a digraph, nevertheless, it is recommended to be used like any other variable name.

To declare and use a classical unsigned short integer:

```
! uint16:
    uint16 .= 0         \ integer: same kind as number 0
    uint16 >= 0         \ value not negative
    uint16 <= 2^16-1

increment (uint16:x):
    x =+ 1
    :> x
```

The programmer should expect that the programme fails with an assertion violation, because the compiler should silently add an assertion after the first statement to check that `x` still is within the constraints of `uint16:`.

Adding an element at the end of a row thus reads:

```
stack::append (val):
    @[@.last+1] =: val
```

or using `this` for convenience:

```
stack::append (value):
    this =: @
    this[this.last+1] =: value
```

## Weak and strong assertions

Assertions are very important to tell the compiler more about the items, and allow optimisations, as the above assertion tells the compiler that x is an integer with at most 4 decimal places, so the compiler can generate standard integer instructions.

There are weak and strong assertions, the latter written with two exclamation marks (`!!`).

Weak assertions can be disabled as runtime checks to increase efficiency, but likewise the probability of undetected errors. Strong assertions can not.

Assertions are used by the compiler to optimize code, thus, they should be used as early and frequent as possible.

If an advanced compiler can ensure that an assertion is not violated, it will not result in code, e.g.:

```
x =: 1
! x = 1
```

If a violation is detected at compile time, a warning may be issued.

If you are — unwisely — concerned with run time efficiency, you should use weak assertions and disable them after thorough testing[18].

In particular, these should be placed as the first lines of a function body:

```
gcd (x,y):
  ! x > 0           \ x must be a positive integer
  gcd -5 y          \ may generate a compiler warning
```

# 5. Functions

## 5.1. Function definition templates

A function definition template has a colon (`:`, or the digraphs `:-` and `:+`, see below) as the last character of the line. Functions are global to the compilation unit, and thus start in column one.

The template is a sequence of words or parameters, the words separated by blank space, and the parameters are enclosed in parenthesis:

```
clip string (str) to the end of string (tgt):
```

Thus, only blanks (and tabs) separating words are relevant, and counted as one blank. Also, as words must start with a letter and contain only letters and numbers, other characters except letters, digits and parenthesis plus the trailing colon are not allowed in a function template. The underscore character is not needed and thus is not considered a letter. However, small and capital letters are fully distinguished.

The templates must obey the following test using the template string:

- replace all white space by a single blank
- replace all parameter positions by the digraph `()`
- terminate each with a colon. Then the resulting strings must start with a word and be unique (pairwise unequal). Additionally, it may not have only one word and the terminator (without parameter). Thus, at any position, there may be any number of words, including zero, optionally a parameter indicator, and optionally a template terminator.

The following templates are valid together:

```
anton berta ceasar dora :
anton berta caesar :
anton berta ()     :
anton berta ()     ()   :
anton berta :
anton ()    caesar :
```

```
anton ()    ()     :
anton ()    :
```

See below for details on the matching process.

The different termination tokens are used as follows:

- If the last character of the line is a single colon, the template is defined local to the compilation unit and cannot be used from outside.
- If the last token is the colon-plus digraph (`:+`), the function is visible from outside the compilation unit.
- If the last token is the colon-dash digraph (`:-`), the template is a declaration for a function from another compilation unit to be linked, and no body is possible.

The last version is used if header files (.tufh) are provided for libraries; the library is scanned for `:+`, and each template issued with `:-` instead. A comment block before is also copied to explain the function, so there is no manual maintenance of header necessary. There is also a special include that only scans for exported templates, in cases where the creation of a `.tufh` file is not warranted.

While not absolutely necessary, it is recommended to separate parameters by words, and not only by nothing or white space; otherwise use a list:

```
copy (a) to (b):    \ good
copy (a, b):        \ shorthand for copy (a) (b)
copy (a) (b):       \ avoid unless best to express the intent
```

Function templates without parameters that have only one word are not supported due to their similarity to plain variables. It is more in the spirit of the language to uses two or more words, as in:

```
next input:
    :> read from ...
x =: next input
```

One prominent example is the print function to send a string to the standard (terminal) output:

```
print (s):
    ...
print "Hello world."
```

If the parameters are really just a list of values, use a single parameter that will be accessible as a list similar to a row within a bunch (provided lists are available):

```
gcd of (x) and (y):
  ? y = 0
        :> x
  :> gcd of y and x//y          \ tail recursion should be efficient

gcd of (list):
  \ calculate the greatest common divisor of a list of integers
  r =: 1
  ?# i =: list.scan
        x =: gcd of list[i] and r
  :> r
print (gcd of (12, 39, 72))
```

Within the parenthesis, not only a single variable name may be written, but also a list. The calling items are assigned the list elements, until exhausted; the remaining elements are set to void. If there are more, the last element, becomes a row itself, containing the remaining elements. Correspondingly, if at the calling place, the variable presented is a row, it is expanded.

The body of the function is defined by the following lines, until another function is defined, or a statement starts in column 1.

The parameters are local variables, which are filled with references to the items that are the actual parameters. In case the parameter refers to a bunch, the bunch can be modified, and the modifications become effective for the body of the function as well as for the caller.

A function always returns a reference to an item; this it void by default, if the last statement of the function body had been executed. Otherwise, the return operator `:>` must be used like in:

```
greatest common divisor of (x) and (y):
    ? x < y
        :> greatest common divisor of y and x
    ?* y > 0
        z =: x %%  y
        x =: y
        y =: z
    :> x
```

There is no restriction on the kind of return values, as these are references, but it is recommended to use the same kind (except void) in all cases, as otherwise the caller might need to check the kind before further processing. Of course, a fault item is the third commonly used kind.

Any variable introduced within the body is a local variable, implicitly initialized with the void reference. Thus, within a loop, it can be checked for void to change the flow if it is already set, or not.

Global variables are denoted by a dollar sign (`$`), followed by the name of the variable. The parameter mechanism is very efficient, as only references (pointers) are used for all items, so there is no reason to use a global variable for efficiency reasons.

There is no declaration of a variable as global because it:

- makes the use clumsy
- avoids trouble by just declaring a variable global
- allows search for the use of a given globals variable by simple find

Note that a global variable is initially void and must be set to void again before programme termination; otherwise a memory leak is signalled.

### Export and import of functions

Normally, functions are local to the file in which they occur. To make them accessible outside, i.e. export them, use the digraph `:+` instead of the single colon at the end:

```
convert (x) to something :+
    ...
```

Alternatively, the template may end in `:-`, in which case no body is expected; this could be regarded as a declaration only, and can occur several times and in addition to the actual function definition with a trailing single colon or the export digraph `:+`. It is used to define the prototypes for library functions, without appending the source code (which is impossible for some basic library functions anyhow).

If functions local to another function are desired, they can be defined using the usual block rules. Only functions local to a function are permitted, not functions local to any other kind of indentation block; i.e. they must begin in the column for the top statements of the function body. This feature is probably not implemented in many compilers.

### Function aliases

An alias for a function can be defined using the `:=` digraph to map the new to the old function:

```
Zeichenkette bis (i) von (s) := clip string (s) upto (i)
```

## 5.2. Function call

In general, a function is called if its template pattern matches:

```
sqrt (x):
    ...
r =: sqrt 2
move (a) to (b):
    ...
move 'aa' to s
```

Matching (part of) an input line is done in a straightforward scan without backtracking, i.e. alternatives are not probed.

In particular:

- If a template has the same word as in the source line, both are advanced, and the scan continues.
- Otherwise, if there is a parameter position in the template, the source line is scanned for an expression. If one is found, it is noted as a parameter value, and the scan continues behind that term and with the next position in the template.
- If there is no parameter or no term matched, but the end of a template reached, the match has succeeded.

A conflict arises if a variable name occurs in a position where it is also a template word. Assume the following two templates and statements:

```
print partial () :
print () nl :
\ use example:
partial =: 1
nl := 2
print partial nl
\ was this meant:
print (partial) nl
\ or this:
print partial (nl)
```

To avoid such ambiguities, it is checked during the match process if a template word at a parameter position is used as a variable elsewhere in the function body. If that is the case, the line is rejected as an error, and the programmer must enclose the variable in parenthesis, as in the lines following the unclear one in the above example.

The compiler has an option to show all these potentially conflicting words and the templates affected.

Template matching has highest priority in scanning an expression, but once an expression scan is started, it continues as far as possible, including function template matching:

```
x =: sqrt 2 + 5     \ same as x =: sqrt (2 + 5)
y =: sqrt sqrt 16   \ treated as y =: sqrt (sqrt 16)
```

Using the first word of a function template as a variable name is is also used as a variable name:

```
sqrt () :
sqrt =: 7
x =: sqrt + 5
```

the expression used as a parameter may not contain In particular, function calls may be used without the expression scan excludes If a parameter position is the last in a template, any expression is recognised; not parenthesis are required if the expression goes upto the end of the line. , even without parenthesis. As an expression includes a function call, a trailing parameter may be a function call, like in:

```
print (s):
    ... send string s to output
string from code point (n):
    ... create one character string with given UTF character
print string from code point 174
print (string from code point 174)  \ this is what is matched
```

As

So matching is greedy as follows:

- If a word matches a template, is is never treated as term that is a single variable (local or parameter).
- If a parameter is possible, and a term is found, it is used as a parameter, even if , even if the end of the template is possible.

- If the parameter position is the last one of template, the line is scanned as far as an expression can be recognised.

If the parameter position is not the last one, only a term is recognised as parameter. A term is:

- a constant (number, string, `?+`, `?-`, or `()`)
- a variable name, optionally followed by an attribute, a row or array index.
- a special variable, i.e. `$#` or a row or array generator
- any expression, enclosed in parenthesis

Under the following templates, the lines match as follows:

```
anton berta   ()     :
    ...
anton ()      caesar :
    ...
calling body:
    anton berta   caesar    \ matches the first one, caesar is a variable
    anton hugo    caesar    \ matches the second one
    anton (berta) caesar    \ force matching the second one
```

If, in the first example line, `berta` is a parameter or local variable, the compiler should issue a warning.

While it would be possible for the compiler to follow a complex algorithm for template matching, this is not true for the programmer. The very strict matching above helps to avoid surprises because the programmer assume a different matching. Also, the compiler would have to undo any variable definitions if a valid term or expression would be discarded during backtracking.

Although in most cases, the templates do differ before the end, it is allowed to have two templates, where one is a prefix of the second:

```
print (x):
    ....
print (x) nonl:
    ....
print 'x' _ nonl            \ matches the first
print 'x' nonl              \ matches the second
print 'x' _ nonl nonl       \ no match, rejected
print ('x' _ nonl) nonl     \ matches the first
```

In the first example, a full expression is allowed for the last parameter, so the concatenation of the string `x` and the contents of the variable `nonl` is taken as the only argument. As a full expression is only allowed as the terminating parameter, and probed only as a last resort, the third example is not a match, because a term is matched for the second template, and then the scan stopped because the underscore is not a word On the other hand, in the third example, the string `x` is a good term, and there is no operator to make it an expression as would be allowed for the first template. Moreover,

This means that the

If the last position of a template is a parameter position (and no word matched), the source line is scanned for an arbitrary expression, like one on the right hand side of an assignment.

but this is also the end of a (short) template, - Otherwise, if the template is complete, the match is successful. - Otherwise, if the template has a parameter instead of a word, a simple expression is expected. If found, its value is the parameter value. - Otherwise, the match fails.

A simple expression is a constant, a variable name, an item attribute or indexed value, or an arbitrary expression in parenthesis. While it would be possible for a compiler to match complex expressions, including function calls, for the benefit of the human reader, a complex expression has to be grouped by parenthesis, in particular function calls.

This general rule has an exception: If a function template matched so far and has a parameter as last element, an arbitrary expression, including a function call, is matched.

Thus, it may be written

```
x =: sqrt sqrt 16
```

```
! x = 2
```

While not very common, it is quite possible that there exist two templates starting with the same sequence of words and parameters, but differing in that first one has a parameter, and the second one a word. In this case, the word has precedence over the parameter; otherwise the word would never be matched and the template useless. However, this might be ambiguous if the word is also used as a local variable. Consider the two templates:

```
get line from (fileid):
get line from stdin:
```

and the program

```
stdin =: new stream reading file '/tmp/xx'
ln =: get line from stdin       \ warning issued
ln =: get line form (stdin)     \ ok
```

To avoid subtle runtime errors, a word that is matched instead of a parameter, may not be used as a local variable or parameter.

Note that the function templates may not be ambiguous, i.e. have not the same chain of words and parameters when of equal length.

There is no forward declaration, even if this requires another pass through the input.

If a single argument is set in parenthesis, they are redundant:

```
r =: sqrt(2)
! r = sqrt 2
```

To supply a list with one element, use a trailing comma:

```
r =: xxx(2,)
```

If the template uses a list, the callee may use a list too:

```
gcd (x,y):
    ...
gcd (15,28)
```

A statement or an expression is first parsed into a list of operators, string or number constants, subexpressions (enclosed in matched parenthesis), and remaining words. Whitespace may be used freely to separate parts of the expression, and is otherwise ignored (except in string constants).

Depending on the precedence and arity of the operators, the row is converted to a tree, where the leafs are rows of constants, subexpressions and other words.

If a row is a subexpression, it is evaluated as an expression.

If a row starts with a constant, it must be the only element.

Otherwise, the row of other words, constants and subexpressions is matched against the defined function patterns.

Then, it is checked, if it is a looper, a guard, or a function return. Then, the rest of the line is evaluated as an expression, see below.

Otherwise, it is checked, if it is an assignment, by searching for an equals sign. The first one found defines the left hand side of the assignment, which is then checked. The rest of the line is the right hand side, and evaluated as an expression.

An expression is evaluated as follows: It is parsed into operators, string or number constants, and remaining words. Depending on the priority and arity of the operators, the expression is converted into a tree, where the leafs are sequences of words and constants. If a leaf starts with a constant, it must not contain more words. Otherwise, the leaf is matched to the function templates. Parameters are matched to the longest string possible, provided it is balanced with respect to parenthesis.

An expression is first parsed, resulting in a row of operator symbols, string and number constants, and

other words. Constants next to an operator are bound to that operator.

The remaining strings are then matched to the function templates, and when a match is found, the

Examples:

```
\ expression must be in parenthesis
a = substr otto from n to (i + 1)       \ does not
a = sqrt 4 + 1  = (sqrt 4) + 1
```

If at the place of the parameter a list is given, i.e. expressions within parenthesis, separated by commas, the parameter is set as a list or row, resp. If a single parameter is The row is then passed to the function, which may have code to use it as a row, or as a

## 5.3. Function returns

A function always returns a reference to an item; if none is provided, void is returned.

To return immediately and to supply a return value, the digraph `:>` is used:

```
give me five:
    :> 5
```

In particular if a function provides the return value, it could be remembered as *continue with*:

```
greatest common divisor of (x) and (y):
    ! x > 0, y > 0
    ? x < y
        :> greatest common divisor of y and x
    ? y = 0
        :> x
    :> greatest common divisor of y and (x %% y)
```

The digraph `>>` can be found in old code; no confusion with a shift operator was possible, because it must be first on the line, and shift operators are not (yet) provided anyhow.

Note that the language implementations check that a function call that is not part of an assignment does not throw away values: If the returned value is not void, a fatal runtime error will occur. Of course, if the return is assigned to a variable and overwritten, the value is still lost without notice.

## 5.4. Function references

Bunch functions should be preferred in order to dynamically select functions; however, the function prototypes must be made available to the compiler at the point of call.

To allow more flexible dynamic invocation, in particular at places where callback functions are required, references can also refer to functions.

Function references can be plain references to functions defined via templates, or refer to anonymous functions, see below.

A function reference is created by enclosing the template in backticks (*acute*, `` ` ``):

```
funref =: `(compare () and ()`
bunch.sort =: `sort record (a) less (b)`
sort array x using `compare funky () with ()`

noop =: ``      \ function that immediately returns void
```

Single Words in the parameter parenthesis' are allowed and ignored, so the prototype can be copied verbatim. There are no operators the can be used on a function reference, so the only places to use are the complete right hand side of an assignment and as a parameter in a function call. Of course, the template must match the templates defined.

To call a function reference, the digraph `:(` (colon followed by opening parenthesis) is used, followed by the traditional list of parameters, closed by a normal parenthesis `)`:

```
x = 3 + funcref:(a, b)
? bunch.sortcall:(x, y)
```

If more parameters are supplied than noted in the function reference, a fatal error occurs, as information is lost. If fewer parameters are supplied, the remaining parameters are set to void.

Note that references to functions are not unique, in that each function reference creates a new item. Nevertheless, they may be compared for equality (only), and the comparison is true if both refer to the same function address; if the number of parameters differs, this is an internal error.

Function reference calls may not be available due to missing lists, then use the library functions

```
call function (func) using (parm):-
call function (func) using (parm1) and (parm2):-
```

As an exit function reference is immutable, it does not have fields. Thus, an exit function registration is not totally simple. Any exit function is called only once, if it is registered again, only the parameter reference is updated:

```
on normal exit call (funcref) with (parm):
    funcs =: get row of exit functions      \ always not void
    ?# i =: funcs.scan                      \ scan skips void
        fr =: funcs[i]
        ? fr.funcref = funcref
            fr.parm =: parm
            :>      \ was a redefinition
    fr =: []
    fr.parm =: parm
    fr.funcref =: funcref
    funcs[] =: fr
```

If the main function returns, the functions are called in reverse order:

```
call exit functions:
    funcs =: get row of exit functions
    ?# i =: from funcs.last downto row.first
        fr =: funcs[i]
        ? fr ~= ()
            fr.funcref:(fr.parm)        \ call using parameter
```

Note that this happens only on normal exit, i.e. return from the main function, to avoid exit problems.

Maybe at some later time there will also exit functions for the premature termination.

## 5.5. Anonymous functions

Anonymous functions use the same backtick notation, but start with an opening parenthesis for the — possibly empty — parameter list:

```
give =: `() $glovar =+ 1; :> $glovar`       \ no parameters
inverted =: `(a,b) compare (b) with (a)`    \ reverse parameters
```

No colon is needed after the closing parenthesis, as only a list of words is allowed in between.

The condensed notation is useful here, e.g.

```
max =: `(a,b) ? a>b : :>a;; :>b`
```

Multiline definitions are also possible:

```
max =: `(a,b)
    ? a > b
        :> a
    :> b
    `
```

Note that an anonymous function is just given a hidden name by the compiler; normally it saves no

space or time; it is just a syntactic expansion which is most useful in combination with the condensed notation.

### Row optimisations

As a string contains characters in the strict sense, it may not be (mis-)used to store a row of bytes; if the input is e.g. in UTF-8 code, two or more bytes may denote a single character. Use Chunks of bytes instead.

Basically, each row element is a reference to the respective item; thus, a row of small integers that would fit into a byte will be larger by a significant factor in the order of 10 than in a more tightly machine oriented language like C.

To solve this issue finally, assertions may be used to assert that each element refers e.g. to a small number. In this case, the array with uniform references can be internally optimised, in that an array of bytes is kept contiguous internally and just flagged in the array header correspondingly.

## 5.6. Bunch functions

To bind functions to bunches, two mechanisms are possible:

- bunch field functions: a field can contain a reference to a function
- bunch tagged functions: extends the namespace

The first case has already been explained above.

Tagged bunch functions are common to all bunches with the same tag, similar to what is called *class functions* in object-orientend programming languages, while field functions are individual, not bound to a tagged bunch, and must be assigned to each newly created bunch.

Tagged bunch functions use the `.tag` attribute to give a bunch a (*class*) name.

An example not using bunch functions might read:

```
get att for mybunch (this):
    ! this.tag = 'mybunch'
    :> this.attr
set attr to (val) for mybunch (this):
    ! this.tag = 'mybunch'
    this.attr =: val
\ use the above templates
    mb =: {}
    mb.tag =: 'mybunch'
    set attr to 5 using mybunch mb
    ! 5 = get attr1 using mybunch mb
```

Note that the template match for functions suggests to use phrases and to position the equivalent of the class name at the end, although the traditional way still has its merits:

```
mybunch (this) get attr:
    ...
```

With bunch functions, the code reads:

```
mybunch:: get attr:
    :> @.attr
mybunch:: set attr to (val):
    @.attr =: val
\ use the above
x =: {mybunch}
x::set attr to 5
! x::get attr = 5
```

The double colon `::` in the function template delimits the tag from the rest of the template, which is the core template. Within the body, the scroll symbol (`@`) denotes the extra parameter which refers to the actual (*this*) bunch.

Calling a bunch function also uses the double colon (`::`) after the variable name (not the tag) instead of single colon or a single dot (`.`); the latter is used with a field that contains a function reference.

An extended template scan is used that determines all applicable core templates and their associated tags. If it is only one, it is just used (with the variable `x` as extra parameter). Otherwise, a dynamic determination is generated like in:

```
x @= 'mybunch'
    mybunch x set attr to val
|? x @= 'yourbunch'
    yourbunch x set attr to val
|
    \ assertion violation generated
```

The constraint system may help to find out the tag of a variable, and then omit the above selection code.

A function created as bunch function always starts with an assertion to match the tag (unless optimised out, e.g. by constraints).

Bunch functions can be grouped in a block (which is functionally the same, but requires less typing and is thus less error prone), starting with the tag string in column 1 and a trailing double colon:

```
tag_name::
    get attr:
        :> @.attr
    set attr to (val):
        @.attr =: val
    is attr equal to (val):
        :> @.attr = val
\ using it the above
x =: {tag_name}
x::set attr to 5
! x::is attr equal to 5
```

As an example, calling code for the Cairo graphic library without bunch functions might read:

```
print header (text) using (ct):
    cairo ct set font size 15
    cairo ct select font face 'monospace'
    cairo ct set source r 0 g 0 b 0
    cairo ct move to x 220 y 40
    cairo ct show text text
```

With bunch functions, it is written as:

```
init:
    ct =: cairo context for aSurface  \ factory function
print header (text) using (ct):
    ct:: set font size 15
    ct:: select font face 'monospace'
    ct:: set source r 0 g 0 b 0
    ct:: move to x 220 y 40
    ct:: show text text
```

In this case, the templates for the library binding are eg.:

```
Cairo:: set font size (num):
cairo context for (surface):
```

So the possible tag names are collected from the function templates; the compiler will match the core template `set font size ()` and find out the tags supplied for these templates.

Using the constraint system via an assertion, any ambiguities could be eliminated:

```
print header (text) using (ct):
    ! ct =@ 'org.cairographics'
```

```
    ct@ set font size 15
    ct:: select font face 'monospace'
    ct:: set source r 0 g 0 b 0
    ct:: move to x 220 y 40
    ct:: show text text
```

Instead of writing `ct::` on each line, a block can be used:

```
print header (text) using (ct):
    ct::
        set font size 15
        select font face 'monospace'
        set source r 0 g 0 b 0
        move to x 220 y 40
        show text text
```

As the bunch function mechanism is not bound to a type system, the class of `com_cairographics` functions could be (locally) extended without requiring something like inheritance, by writing:

```
com.cairographics::print header (text):
    set font size 15
    select font face 'monospace'
    set source r 0 g 0 b 0
    move to x 220 y 40
    show text text
```

Of course, an own class could be used, where the bunch contains a field that refers to the cairo item:

```
drawing::initialize:
    @.ct =: cairo context for @.surface
drawing::print header (text):
    @.ct::
        set font size 15
        select font face 'monospace'
        set source r 0 g 0 b 0
        move to x 220 y 40
        show text text
```

The user needs not to know the tags for a library function

Note that the tag name needs not to be known at the place of use, as it is derived from the template declaration (and so normally shown in the library documentation).

The double colon when calling a function is admittedly a but clumsy, but the single colon is used for [Assertions and Constraints][19].

The order of a constraint for the function return value and a bunch function prefix is free, but as the bunch function prefix can be blocked, and a constraint not, usually the former comes first. An example is still missing.

### Constructors and destructors

Constructors and destructors are defined here only for completeness; factory functions should be used instead. In order to indicate that a tagged object with a constructor is created, the syntax to create tagged bunches is extended:

```
x =: {!tag}
y =: [!tag]
```

Alternatively, creating a bunch with a word as a tag could always require a constructor.

Constructors and destructors can be defined by using the digraphs `+:` and `-:` as function templates:

```
! stack:                    \ constraint declaration (optional)
    @.tag = 'stack'
    @ .= []                 \ is a row
```

```
    @.top .= 0              \ has the field top as integer

stack::
    +:
        @.top =: 0          \ make it an integer
    -:
        ! @.top = 0         \ just for demonstration, not sensible
    push (x):               \ the row expands automatically
        @.top =+1
        @[@.top] =: x       \ do not use @[] = @[@.count+1]
    pop:
        ? @.top = 0
            :> ()
        rv =: @[@.top]
        @.top =- 1
        :> rv
    top:
        :> @[@.top]         \ return void if exhausted
    clear:
        @.top =: 0
    copy:                   \ the copy is condensed
        ns =: []
        ns.tag =: @.tag
        ?# x =: @.scan
            ns:push @[x]
        ns.top =: @.top
        :> ns
new stack:                  \ a factory function
    :> [stack]              \ tagged as 'stack', with constructor
\ using the above:
st =: new stack
st:push 5
! st:pop = 5
st =: ()                \ destroy stack
```

Constructors and destructors do not have parameters and must return void, as it is discarded. If it is required to have constructor parameters or to return fault items, separate initialisation and termination function must be provided, often in conjunction with factory functions. Normally, a destructor will contain only assertions to assure that no valuable information is lost.

A small inconvenience is that the user must know whether to use a row or an array in creating a tagged bunch with bunch functions, so factory functions will often be used.

### Factory functions

Factory functions must also be used if parameters would be required for constructors. They can use initialisation functions from the bunch for most of the work.

Such an example is a queue:

```
new queue with (n) entries:
    rv =: [queue]
    rv::setup queue of length n
    :> rv
 queue::
    setup queue of length (n):
        @.in =: 0
        @.out =: 0
        @.filled =: 0   \ allows to use all cells
        @[0] =: 0       \ ensure first index zero
        @[n-1] =: n     \ allocate space
        ! @.count = n
    is empty:
        ? @.filled = 0
            :> ?+
```

```
            :> ?-
        is full:
            ? @.filled = @.count
                :> ?+
            :> ?-
        enqueue (x):
            ? is full       \ template with same tag has priority
                :> new fault item with code 1 and message "Queue full"
            @[@.in] =: x
            @.in =: (@.in+1) %% @.count
            @.filled =+ 1
        dequeue:
            ? is empty
                :> ()
            rv =: @[@.out]
            @.out =: (@.out-1) %% @.count
            @.filled =- 1
            :> rv
    use it:
        q =: new queue of length 100            \ factory function
        ! q::dequeue = ()
        q::enqueue 5
        q::enqueue 'hallo'
        ! q::dequeue 5
        q::enqueue 3.5
```

In general, factory functions should be used instead of creating a tagged bunch with the correct tag, because:

- It must be known if the bunch is a row or an array
- The constructor may not provide enough options
- During initialisation, the state may be incoherent

## 5.7. Forking subprocesses

Nothing special has to be done here; just the two basic library routines

```
process fork
process wait
```

are provided, functioning as in any POSIX environment. Of course, the newer more flexible system calls may be provided by the standard library.

## 5.8. Standard library function examples

Here, only a few functions are sketched; see the separate documentation on the TUF standard library.

### Arithmetic functions for floating point numbers

math sqrt (x):
: provides the square root of a number as floating point number, exact numbers are accepted and converted to float

### Arithmetic functions for integers (and exact numbers)

greatest common divisor of (x) and (y):
: returns the greatest number that divides both, x and y

### String functions

Instead of *substring*, which is regarded as a legacy from one-word function names, the term *clip string* is used.

clip string (s) from (first) to (last):

return a copy of the string between the characters between the integer positions first and last, counted from 1; if first is greater than the length, return the empty string; if last is greater than the length, return upto the end. if last is less first, return the empty string. if first or last are less 1, raise a run time error

clip string (s) from (i) length (l):
:> clip string s from i to (i+l-1)

clip string (s) from (i):
:> clip string s from i to s.length

clip string (s) upto (i):
:> clip string s from 1 to i

split string (s) to row of characters:
returns a row, indexed by 1 upto s.length, where each element refers to a sting containing just the i-th character. Remember that short strings are stored as efficient as integers.

split string (src) by any of (delim):
split `src` into an array of strings using any one of the characters in `delim` as delimiter. If two delimiters follow directly, an empty string is assumed.

split string (src) by many of (delim):
split `src` into an array of strings using as many of the characters from `delim` as delimiters as possible. Thus no empty strings are found. Useful for scanning for white space with e.g. `&tab;`

enumerate string (s) as characters:
returns all characters of s as one-character strings in a loop

enumerate string (s) as integers:
returns all characters of s as integer code numbers in a loop

Because there is no single character type, character classes are better supplied by matching from a starting point (see `man strspn` under Unix); interfaces to obtain character tables, in particular for localized strings, may be added:

count in string (src) any of (accept) starting at (start):
returns the number of characters in the string `src`, starting at `start`, which consist only of characters from `accept`. If none is found or start is larger than src.length, void (not integer 0) is returned.

count in string (src) none of (reject) starting at (start):
returns the number of characters in the string `src`, starting at `start`, which consist not of characters from `reject`. If none is found or start is larger than src.length, void (not integer 0) is returned.

Note that if both functions are called in sequence with unchanged parameters, either both return void, in which case start is greater than the length, or one returns an integer greater zero. In other words, if start is less or equal the string length, and the first returns void, the second will not. So in the following code, the loop will always terminate, as the index is incremented by at least one each iteration.

Using these, the above word split could be written in TUF. If the source starts or ends with delimiters, an empty string[20] is stored as the first resp. last element of the result. Consequently, a string with only delimiters returns a row with two empty strings. Note that the number of elements in the result string is always one more than the number of delimiter chains, thus the index is advanced with every delimiter string.

```
split string (src) to row using delimiters (dlms):
    res =: []
    ? lengthener < 1
        :> res
    idx =: 1
    res[idx] =: ''              \ if starting with delimiters
    start =: 1
    ?* start <= lengthener
        n =: count in string arc any of elms starting at start
        ? n ~= ()               \ delimiter chain found
            start =+ n
            idx =+ 1
```

```
            res[idx] =: ''      \ to be overwritten except at end
        n =: count in string arc none of elms starting at start
        ? n ~= ()               \ non-delimiter chain found
            start =+ n
            res[idx] =: part of string arc from start length n
    :> res
```

More string functions:

split string (s) by string (t):
: ...

split string (s) by reg exp (r):
: ...

match re (r) in string (s):
: :> match reg exp r in s

match reg exp (r) in (s):
: returns () if the match fails; otherwise returns a row with three elements: the string before, the matched string, the string after.

find in string (str) regular expression (re):

split string (str) using re (re):
: returns void if the regular expression is not matched. Otherwise, returns an row with five attributes: - start: integer index of first character that matches - length: integer index of last character - before: string of the part before the match - match: string from the source that matched - length: length of the part that matched If groups were present, the bunch is a row, having pairs of indices (starting at 1) of index of character index and length of each group. Note that the length may be zero, if the empty pattern was matched. first and last character of the group.

position of string (s) in string(t):
: Find first occurrence of s in t; if found, return the index (starting with 1); if not found, return void (not the integer 0).

substitute (r) by (u) in (t) once:
: ? (s, l) =: match r in t t =: substring of t upto s-1  u  substring of t from s+l :> t

enumerate indices of (p) in string (s):
: ni =: $# ? ni = () ni =: 1 ?# i =: from ni to (s.count - l) ? p = part of s from i length l $# =: i+1 :> i $# =: () :> ()

Returned is the number of characters

### Conversion to and from numbers

integer from string (s):

### Formatting

apply (format) to (value):

### File functions

See list of library functions.

### File execution

See list of library functions.

# 6. Various

## System variables and parameters

The runtime system creates an all-global variable `$SYSTEM` (similar to `$COMMON`) with some special

attributes and fields.

The attributes are:

| tag | kind | contents |
|---|---|---|
| .parms | row | parameters as provided by the command line |
| .env | array | environment strings |
| .intmax | integer | maximum (signed) integer value ($2^{63-1}$) for high-speed arithmetic |
| .ratmax | integer | maximum (signed) integer value ($2^{31-1}$) for high-speed rationals (numerator or denominator) |

The fields are:

| tag | kind | contents |
|---|---|---|
| .debug | boolean | debug enable |
| .stats | boolean | print runtime statistics at end |

There are also several functions to access these values.

The runtime uses some command-line arguments and removes it from the originally given row when calling `main(parms)`; the attribute `$SYSTEM.parms` gives the original ones. As this time of writing, the following were used:

```
--debug             switch debug on
--runtimestats      print runtime statistics at end
```

Some of these can also be set and obtained by function:

```
debug enable:
    $SYSTEM.debug =: ?+
debug disable:
    $SYSTEM.debug =: ?-
runtime stats enable:
    $SSTEM.runtimestats =: ?+
```

# Program and shell invocation

There are library functions to execute a shell command; these can be disabled if the environment variable `$NO_SHELL` is set upon invocation to any string; its effect cannot be reverted. Attempts to use the functions if disabled is a fatal runtime error

The exchange of the current program image with another one (`execl`, `execv`) is inhibited by `$NO_EXEC`, which also inhibits shell invocation. Any attempt to use is a fatal runtime error.

There is no use to hide these environment variables from malicious programs; if a malevolent user can start a program once, he can do so twice.

# Printing

There are functions to open a stream to `stdout` and `stderr`, resp., so that standard input-output functions can be used for console output and the like.

Because this is not very handy, a group of functions provide a shorter way and automatic conversion of numbers to strings:

To send a line to standard output, the function `print (x)` can be used. Strings are sent unchanged; numbers and booleans are converted to strings, all without leading or trailing blanks.

All others are rejected as a fatal runtime error.

The conversion uses the same format as the string catenation operator _, it is quite helpful to compose an output line this way.

Future versions might support lists, so that commas can then be used instead of underscores.

If the output should be the standard error stream, use `error print (x)` instead.

As a debugging aid, the function `debug print (x)` suppresses the output unless the debug flag is set. Note however, that the function is called normally, including evaluation of argument(s), so inside heavily used loops its better to use:

```
? is debug enabled
    print ....
```

To aid debugging, any item can be made legible (sort of) and written to standard error, mostly in several lines, by:

```
debug dump (item)
```

Additionally, `debug print this location` prints the current source code function template and line number (within the file), and `debug print stack trace` a stack trace.

As file output normally adds a newline at the end, the same is true for the print variants that are convenient to provide console output. So in general, a complex line should be composed by string catenation, i.e. _ and =_, and then emitted.

This is sometimes cumbersome; the often used solution that the programmer must provide newline characters explicitly is not systematic, thus error prone and also onerous in many situations.

The functions to print without trailing linefeed come have an old and new version:

- `print partial (x)` \ old version, deprecated
- `print (x) nonl`

and similar for the other print variants.

If a carriage return is desired, it must be written by `&cr;`.

# Tail recursion

Mathematicians are much trained to reduce a problem to a chain of already solved problems, and thus often use recursion.

In particular, a fully recursive solution for the greatest common divisor of two numbers is:

```
gcd (x, y):
  ? y = 0               \ mathematical definition of gcd
    :> x
  :> gcd (y, x mod y)   \ note that r is the mathematical modulus
```

There is no need to do a real recursion and to save x and y when calling gcd recursively, because the values are discarded immediately on return. this situation, also called *tail recursion*, allows just to set the parameters anew and start over as a loop, effectively creating

```
gcd (x,y):
    ! x >= 0, y >= 0    \ must be non-negative integers
    ? y > x
        :> gcd (y, x)   \ could also swap x and y
    ! x >= y
    ?* y > 1
        z = x%%y
        x = y
        y = z
    :> x
```

The programmer should assume that the compiler will do this kind of optimisation, an do not hesitate to use tail-recursion.

Tail recursion is relatively easy to detect, but difficult to do in C, thus it may not be present in most compilers.

# 7. Libraries and Modules

## Libraries

Libraries are essential for the use of a programming language.

Some very basic libraries for file input-output etc are part of the core language. They use the special `system` item, of which only a reference may be passed and copied; changes are by system functions only. Core library bindings use a set of a few (about 10 of 255) subkind codes to protect against improper used.

The remaining subkind codes are dynamically shared by other bindings. The runtime system maintains mapping of identification strings, like `de.glaschick.mylib` or `org.cairographics.surface`:

```
byte map_system_foreign_subcode(const char* id);
```

It is just a table of strings; if the string has already be used, the code byte will be returned; if not, it will be remembered and a not yet used code returned. An entry is never deleted until the final end of the program. This allows about 200 foreign libraries used by a single program to coexist, which should be sufficient. {There are 16 bits on a 32-bit system assigned to kind and subkind, using 8-bit bytes, leaving room it really this language is so prominently used that this becomes a problem.).

## Modules

Modularisation in TUF-PL is — inherited from C — tightly bound to source code files; a function can be internal or visible from the outside.

To inform about functions in other modules, there are two methods:

- A list of prototypes ending with `:-` is included with the normal include (`\+` in first column); similar to commonly used header files in C and other programming languages.
- Using the digraph `\^` instead, the the included file is scanned for exported function prototypes only, i.e. starting in colum 1 and ending with `:+`; these are treated as ending in `:-` instead.

If the filename starts with a slash or with `./`, it is used as is, i.e. as absolute or in the current directory. Otherwise, i.e. starting with a letter, digit etc, a standard search path is used. The syntax `~user/...` and `~/...` may be used too.

No variables may be shared by different source files, i.e. modules.

In either case, the name of an already compiled library to be included by the linker may be indicated by the pragma `\$`, mostly used inside the files included by `\^`. This may not be possible on some systems; it depends on the linker calling mechanism.

In particular, the line is copied to the C source code, with `\$` replaced by `//$$`. The script to compile and link the main function will just search for lines beginning with `//$$`, and supplies the rest as additional linker options. For TUF files, this may be the corresponding object file, e.g. *\$ mylib.tuf.o*. In particular, interfaces written in C might require extra libraries; e.g. the source file TUFPL crypt.c may not only need its object file, but also `-lcrypt`. Thus, the source code shall contain

```
//$$ -lcrypt
//$$ TUFPL_crypt.o
```

Of course, the header extraction program `defextract` has to copy both forms to the header file.

## External C functions

As TUFPL compiles the source into a C (or C++) function, linking with functions written in C is possible, provided that these functions behave like a TUF-PL function. This is fairly easy; just a bouquet of auxiliary C functions must be used to obtain the content of items, create new items, etc. To allow functions to pass memory to later calls via items, a special kind of *opaque* or *foreign* item is provided, that may be freely passed and copied by TUF-PL code. In order to …

# Modules

TUF-PL has been created as a precompiler for the *C* programming language. Thus, a module is a compilation unit that will result in a binary machine code object, that can be arranged in a library etc.

While in C, every function is exported, unless marked as local, TUF-PL treats every function as local to the compilation unit, unless marked as exported.

There are several flavours of modularisation:

Pure binary:
: The main programme is compiled, and modules exist as binary machine code objects, bound together by a linker, before the main programme is executed as binary machine programme.

Purely interpreted:
: The main programme is interpreted, all modules are read as source code and treated as included verbatim, except some basic library routines, that are resolved by the interpreter

Mixed:
: The main programme is interpreted, and modules exist in a precompiled form.

# Pure binary modules

In its simplest form, the file to be imported is just scanned for exported function templates and import pragmas. No code is generated; just the proper declarations for the C code issued. It is left to the development environment (work bench) to compile and link the binary objects accordingly.

In order to assist the work bench, the path names that are imported are printed by the precompiler, which might be scanned and used to prepare link commands.

The imported file may well be the sourcecode; while in import mode, the precompiler will just scan for function templates marked exportable and import pragmas. No detached header files are necessary, a mostly efficient enough solution for compilation modules in a project.

In case of libraries, the files imported may well contain only the the templates, thus be equivalent to header files used elsewhere. This may be a single header file collected from all the modules in the library, using TUF-PL syntax, together with the `\$` pragma to indicate the path to the library, which is printed too.

If a library — including the runtime system — is not written in TUF-PL, a header file should be created because - packaging is required anyhow - the header reflects the library

Recommended extensions are `.tuf` for TUF-PL source code, and `.tufh` for header files that are not meant to be compiled to object code, in particular to provide a library interface.

# Message translation

Strings in (double) quotes are translated by the runtime system, if a translation mechanism is provided; otherwise, they are used as-is.

The elementary mechanism is to set a translation array:

```
set string translation array (ary)
```

The creation of the array may be done by array literals or reading configuration files. The parameter must either be an array or void, in which case the translation is suspended.

Currently, string translation takes place if the string literal is used in the program flow. This is rather efficient, as just another item reference is used from the translation array.

However, the value of a string literal depends on the situation when it was created, i.e. assigned or otherwise used, as in the following example:

```
\ translation is done when the literal is dynamically encountered
arg =: "argument string"
print arg
```

```
set string translation array cv
print "argument string"
print arg
```

which will — somewhat unexpectedly — print:

```
argument string
Uebersetzte Zeichenkette
argument string
```

It might be less error-prone if the translation takes place when the string contents is finally used, but requires more processor time.

Instead of a translation array, an interface to *GNU gettext()* may be provided, possibly by allowing to register a callback for translation.

# 8. Object oriented programming

## Introduction

There is a saying that good programs can be written in any language, but in some it is easier. Similar, object oriented programming can be done in any language, but some make it easier. The reverse is also true: bad programs can be written in any programming language, including those that enforce object oriented programming.

TUF-PL is hopefully a language that allows good programmers to write good programs, provided that the programmer does not assume that the language will automagically correct his errors.

Instead of complex rules governing many features, a simple language with a few rules, where complex issues are to be code explicitly, is deemed more secure: A single sharp knife is in several cases better than a set of specialized cutting tools.

If someone ever has written documentation for a project that uses OO language with classes and inheritance including in-depth reasoning why a certain interface is correct and works, will know that this is as complex as with a simpler language. Assuming the the OO language needs less documentation is a failure that might be the reason why software still needs frequent updates.

That strong typing is not necessary show languages like Python and Ruby, and TUF-PL is a language that supports both via the extended assertions.

Furthermore, object oriented design is what it says, a design method. Thus, the major questions are:

- can any object oriented design be expressed clearly?
- can errors be detected early?

The most advanced answer to the latter question is to use design tools and leave the mapping to a particular language to the tool, avoiding a lot of errors early. If the first question is positively answered for TUF-PL, the second is trivially true too.

For implementing a design that used the object oriented approach, none of the special features described next are really necessary. Bunches can be used to represent objects, and assertions can be used to not only write down, but also verify pre- and postconditions on the objects. It admittedly requires discipline, which is quite possible compared to the discipline and careful planning hardware engineers must obey.

As bunches provide:

- numerically and associative indexed aggregation
- fields, called attributes, with any contents

the first question is assumed to be true.

Not that constraints are not applied to bunches that are locked.

## Features for object oriented programming

Many object oriented languages include data abstraction, because it fits well into the the idea of objects to which the relevant operations are assigned.

Data abstraction uses functions to get information from objects and to manipulate objects. This is clearly possible in TUF-PL.

In OO languages, the functions defined for the object are often prefixed with the name of the object. This allows shorter functions names, which is desirable in all commonly used OO languages as the function name is always a single word. With the much longer and flexible function templates in TUF, the object class can be used as as part of the function template. This can be studied with the templates for the standard library.

In order to allow either the compiler or the function to check that the bunch to process is not corrupt, two very similar mechanisms can be used:

- the `.tag` attribute can be used to build tagged classes of bunches
- constraints, i.e. assertion declarations can be used, which may check the `.tag` attributes, among others.

Besides these basic ones, we have:

A function template may start with a sequence of words, the tag (*class name*) of the function, followed by a double colon:

```
aClass::template to do (x) and (y) :
    \ code for the class member
your class::template to do (x) and (y) :
    \ code for the class member
de.glaschick.aClass::template to do (x) and (y) :
    \ code for the class member
de_glaschick_bClass::template to do (x) and (y) :
    \ code for the class member
```

Although the tags are strings, no (single) quotes are neither required nor allowed; just sequences of words separated by points or blank space.

When setting the `.tag` attribute, quotes must always be used to form a string, even for a single word:

```
x := {}
x.tag =: 'aClass'
```

Note that setting a tag can only be done once.

A bunch of a certain class is created using the class name inside the branches:

```
row =: {'aClass'}
yourArray =: {'your class'}
array =: ['de.glaschick.aClass']

row::template to do 1 and 2     \ calling a class function
```

The tag name inside the braces and brackets must be a string literal or a variable referring to a string item; thus the quotes are required if a literal is used. While this seems inconvenient at first, in many cases a tagged bunch is created by a factory function; so the use of literals is not so often as might be expected.

Inheritance by matching the prefix might be defined in far future.

### Example

The following are the templates for the *Femto Data-Base*:

```
open fDB(pathname):
    returns an fDB handle (object) for an existing database
new fBD(pathname):
    creates a new database and returns the fDB handle (object)
```

```
fDB::
    get information:
        returns also the fieldname record

    close:
        closes the associate file

    get single record at key (key):
        if the key does not exist, void is retuned

    give records after (key):
        Starting with key, records in ascending order are supplied.
        If exhausted, void is returned.
        If key is void, the first is returned.

    count records between (begin) and (end):
        Returns the number of records between (begin) and (end),
        both included, if they exist.
        If (begin) is void, it is before the first.
        If (end) is void, it is the last.

    add (newrec):
        The record with the key in (newrec) is added.

    delete (oldrec):

    update (newrec):

\ usage:
xfdb =: new fDB('./new_fDB.fdb')
! xfdb.tag = 'fDB'          \ (single) quotes are required
info =: xfdb::get information
xfdb::close
```

## Questions and Answers

In the following questions, the term *support* always means that there are special means in the language to support.

Does TUF support class variables?
: No. Global variables are sufficient for this, as they are global only to the current compilation module, thus no collision with globals in other modules can occur. Checking the use of global variables within a module is not very costly.

Does TUF support class methods?
: No. Class methods are considered a misuse of the class concept, in order to avoid name spaces or other means to support modules. E.g., file handling is not a class, but a module. A file object can be created, and then all the methods can be applied. Functions that do not need objects are not class functions, but functions within the module.

Does TUF-PL support inheritance?
: No, not directly. However, any tagged bunch may use the attribute `.super` to link to a parent. But, in contrast to normal inheritance, all inherited functions must be written down using the corresponding function from `.super`. This makes the dependence for features of the parent visible. An this allows to simply *override* the parent's function, and even to correct interface modifications.

Does TUF support singletons?
: No. Global variables are a simple means to share data between functions in a module. Note that singletons are normally deemed necessary if a resource is unique and its used must be synchronised. So use a global variable to save the reference to a bunch describing that resource, and have all functions use this resource. Release of this resource must be programmed explicitly for good reasons, because it is easy to forget the often complex rules that allow automatic release, and often the assumption that the rules are fail-safe is wrong.

What about constructors and destructors?

Use factory functions instead (see above)

# 9. Examples

More examples can be found online at http://rclab.de/TUF-Online.

## Hello world

The famous example:

```
main (parms):+
    print "Hello, world!"
```

## Table of squares

Early computers often used this a first test:

```
main (parms):+
    ?# i =: from 1 upto 100
        print i __ i*i
```

No multiplication is necessary, as $(x+1)^2 = x^2 + 2x + 1$, or better $x^2 = (x-1)^2 + 2x - 1$:

```
main (parms):+
    x2 =: 0
    ?# x =: from 1 upto integer from string parms[1] else 100
        x2 =+ x + x - 1
        print x __ x2
```

## Approximate square root

Calculate the square root by Heron's method:

```
sqrt (a):
    ? a < 1.0
      :> 1.0 / (sqrt 1.0/a)     \ tail recursion
    x =: a
    y =: x * 1.01               \ thus y > x
    ?* y > x                    \ monotonically falling until sqrt found
       y =: x
       x =: (x + a/x) / 2.0
    :> x
```

## Greatest common divisor

```
greatest common divisor of (x) and (y):
    ! x > 0, y > 0           \ implies integers
    ? x < y
        :> greatest common divisor of y and x
    ?* y > 0
        z =: x // y
        x =: y
        y =: z
    :> x
```

Using tail recursion:

```
greatest common divisor of (x) and (y):
    ! x > 0, y > 0
    ? x < y
        :> greatest common divisor of y and x
    ? y = 0
```

```
        :> x
    :> greatest common divisor of y and (x // y)
```

## Linked List

Many scans are predefined; here comes a scan that goes through a linked list:

```
traverse linked list (first):
   ? $# = ()             \ if initial call
        $# =: first      \ start at the beginning
    |
        $# =: $#.next    \ else advance, may be void
    :> $#

create linked list from (r):
    ? r.count = 0
        :> ()
    x =: []
    x.val =: r[r.first]
    x.next =: ()
    ?# i =: from r.first upto r.last
        n =: []
        n.val =: r[i]
        n.next =: ()
        x.next =: n
        x =: n
    :> x
r =: 3, 13, 7, 11, 5, 9
l =: create linked list from r
?# v =: traverse linked list (l)
       print v.val
\ not much more complicated if used directly:
v =: l
?* v ~= ()
    print v.val
    v =: v.next

\ but we may want to filter the results:
traverse linked list from (first) below (limit):
    hic = $#
    ? hic = ()             \ start with first
        hic =: first
    |
        hic =: hic.next
    ?* hic ~= () & hic.val > limit
        hic =: hic.next
   $# = hic               \ done if void, also if first = ()
    :> hic
\ and use it like this
?# v =: traverse linked list from l below 10
    print v.val
```

# 10. Features not (yet) approved

This section collects language variants that have not yet been accepted, but that could be integrated seamlessly if so desired and considered useful.

Some of the following texts are not up-to-date and are to be thoroughly revised.

## Sequences

Primarily for interactive use and for short anonymous functions, a condensed notation allows two or

more statements on one line. (This feature is still not implemented.) A semicolon (`;`) is treated like an end of line, with the following statement in the same block (same indentation level):

```
x =: 0; y =: 0
tok =: toks[ti]; ti =+ 1
```

If the line starts with a question mark for a guard (if or loop), a single colon opens a dependent block like a line end:

```
? a = (): a =: []                   \ set empty row if not yet set
? i < 0: ?^                         \ conditional repeat loop
? i > 0: ?>                         \ conditional terminate loop
?* x.next ~= (): x =: x.next        \ find last in chain
k =: []; ?# i =: from 1 to 10: k[i] =: 0    \ fill array with integers
? x.isfault: x.checked =+ 1; :> x       \ propagate fault up
? x > y: z =: x ?~ z =: y           \ determine maximum
? x > y: z =: x; | z =: y           \ same, observe ; before |
```

The *if-not* (*else*) token (`?~` or `|`) closes the current block and opens a new block; note the semicolon before the vertical bar, as otherwise it would be the boolean `or`-operator. The colon notation is restricted to a guard; it is not possible for any notation having a dependent block.

Two lines are also possible if an alternative is present:

```
? x > y: z =: x
| z =: y
```

A double semicolon is reserved for block close, but should be avoided for clarity. For complex situations, better use a function instead.

The condensed notation is useful in short anonymous functions:

```
max =: `(a,b) ? a > b: :>a; | :>b`
```

Because the dependent statements are not limited to assignments, two conditional statements may be nested, e.g.

```
? x ~= (): ? x.attr = val: :> x
```

Also, an indented block can be used as in

```
? x ~= (): ? x.attr = val
    ...
    :> x
```

Note the absence of a trailing colon in the second part. Indentation of the dependent block is checked against the indentation of the primary statement.

In order to have the feature defined, not necessarily implemented, conditional expressions may be used in a form similar to conditional expressions:

```
x =: ?( discriminant ?: trueval ?~ falseval ?)
```

to avoid writing down the variable twice, as in:

```
? logical expression
    x =: trueval
|
    x =: falseval
```

The discriminant is a boolean expression, and `trueval` and `falseval` represent any expression that would be allowed at the right hand side of an assignment. Additionally, the expression may be preceded by simple statements, separated by semicolons.

### Integer exponentiation

Supply integer exponentiation by *square and multiply*.

## Slices

Currently, no slices are included in the language.

However, to have the language ready, and check that they would fit in, here is what they would look like.

First, the index of a row may be a list, extending the list to the row cells:

```
r[1, 2, 3, 4, 5] =: 1, 2, 3, 4, 5
a, b, c =: r[5, 6, 7]
```

Or using lists of strings:

```
r{'a', 'b', 'c'} =: 1, 2, 3
a, b, c =: 'a', 'b', 'c'
x, y, z =: r{a, b, c}
! y = 2
```

Using the slice symbol, two fullstops, a range of integers is a list:

```
a, b, c =: 1..3
```

While it would be tempting to allow

```
r[1..] =: 1, 2, 3, 5, 7, 11, 13, 19
a, b, c, d, e, f =: 1..
and leave the upper limit out, this would define a unary postfix
operator, and is thus not considered.
```

Strings considered lists of characters, the slice would also help there:

```
s =: 'The quick brown fox'
t =: s[5..9]
! t = 'quick'
```

### List and array generators

The use of lists inside bunch generators to create rows or arrays filled with literal values, as in

```
row =: [9, 'y', sqrt 2]
array =: {a=:9, b=:"x", c=:3.0}
```

is not needed for rows if lists are provided, as just the square brackets are to be dropped:

```
row =: 9, 'y', sqrt 2
```

The same for arrays can be done via functions if lists are supported:

```
array =: array from list ('a', 9), ('b',"x"), ('c', 3.0)
array from list (list):
    rv =: {list.count}
    ?# le =: list.scan
        rv[le[1]] =: le[2]
    :> rv
```

In general, just strings are parsed, as in the following library routine, which does not allow colons and commas in strings, etc:

```
array =: array from string 'a:9, b:"x", c:3.0'
array from string (str):
    \ strings may not contain commas
    list =: split string str by any of ','
    rv =: {}
    ?# le =: list.scan
        lev =: split string list[le] by any of ':'
        arg =: lev[1]
```

```
            val =: lev[2]
            arg =: replace each ' ' by '' in string arg
            val =: clip string val without leading white space
            val =: clip string val without trailing white space
            res =: tmp
            tmp =: integer from string val      \ wrapper that returns null
            ? is tmp integer
                res =: tmp
            |
                tmp =: float from string val
                ? tmp.isfault
                    tmp.checked =+ 1
                ? is tmp float
                    res =: tmp
            ? val[1] = '"' | val[1] = "'"
                res =: clip string val from 2 upto -2
            rv{arg} =: res
        :> rv
```

Admittedly, the row literals are a bit easier to read as in:

```
    array =: {
            a =: 9,
            b =: "x",
            c =: sqrt 3
            }
    \ basic form:
    array =: {}
    array['a'] =: 9
    array['b'] =: "x"
    array['c'] =: sqrt 3
```

Replacing the assignment digraph by a colon is not appreciated, as it could regarded as constraint, even if constraints are not sensible inside a bunch generator.

**Attribute change callbacks**

In object-oriented languages, object attributes are normally protected and changed by the tedious getter/setter object functions, which often require many lines of code even if the values are just stored, and make the code clumsy with all the function calls. As this cannot be changed later, it is necessary to do so for all field values, even if there is not yet any need to do so.

In TUFPL, an attribute of a bunch can be associated with a guard function, that is called if a value is inquired or changed. This has the advantage of very clear code; because setting is just an assignment, not a function call. There is no need to precautionary use getter/setter functions in case a guard function might be required later. There is clearly a — presumably small — penalty in execution size and speed[21].

Instead of

```
    obj.set_width(5)
    x = obj.get_width()
```

this is just a normal assignment in TUFPL:

```
    obj.width =: 5
    x =: obj.width
```

To associate a getter or setter function, a special assignment is used:

```
    x.width =< `set width of (bunch) to (new)`
    x.width => `get width of (bunch)`
```

Using tagged bunches in an object oriented manner:

```
    a_Class::
```

```
        get width:
            :> @.width
        set width (val):
            @.width =: val
        +:
            \ anything to be done if the bunch is created
            @.width =< `.set width ()`
            @.width => `.get width`
        -:
        \ code if resources have to be freed

    x =: {a_Class}
```

Note that many cases where setter functions are used to enforce integrity can be solved with constraints.

For a setter function, the value returned is set to the attribute; for a getter function, the value returned is used instead of the *true* value of the attribute.

Not that comparison is done with >= and <=, so there is no collision[22].

This feature is very handy for GUI interfaces, as a GUI object is just changed.

A getter or setter function can return a fault item if there is a recoverable error encountered, which is then stored in the attribute in case of the setter function, and given back to the caller in case of getter functions.

Note that e.g. in C++, the compiler may optimize a getter or setter function to inline code, in particular if just a field value is obtained or changed. But in these cases, there is no function associated, and the relative efficiency is the same, without stressing compiler optimization.

**Overriding bunch element access**

A bunch has two attributes, `.getOverride` and `.setOverride`, that can each hold by a reference to a function. The `.getOverride` function is called whenever the row or array is accessed, i.e. the form `row[index]` or `array{index}` is found within an expression, and should return a value, while the `.setOverride` function is called if the form is the target of an assignment. When such an override function is called, a flag in the bunch is set to disable the override function until it returns, so that the override function can use normal direct access.

This feature has still to be defined.

**Regular expressions**

Support for regular expressions is an optional module; using the standard string functions is often clearer.

Regular expressions are written as (non-localised) strings. All library functions which use regular expressions use a common cache to compile the regular expressions on demand.

There is no need for a special match operator symbol, as the standard functions all return a void reference if they fail the match:

```
    ?* find re '[a-z][0-9a-z]*' in s       \ returned values ignored
      do something
```

Most often, in case of a match, the position and length is needed and returned as a bunch with two fields:

```
    ?# x =: find re 'a[0-9]+b' in s
      ? x.start > 0 & x.length > 0
        x =: clip string s from s length l
```

If list assignments are implemented, the example might read:

```
    ?# b, l =: find re "a[0-9]+b" in s
```

```
? b > 0 & l > 0
  x =: clip string s from b length l
```

**Enumerated integers**

No language support is provided to assign a range of numbers to a list of names automatically, like the *enum* in C. This is assumed a legacy from early computers where it was common to assign integers as codes for smaller and quicker programs. Using (short) strings instead, the loss in speed or size is practically none. Moreover, such a feature would only be useful within a compilation unit, and thus there is no language feature.

If it is desired to use names instead of numerical constants, use Named Literals instead.

**List items**

A special form of lightweight rows could be used for lists, in order to save space and time. These list items would have the following differences:

- the size is fixed at creation
- indexed by 1, 2 etc
- no fields, no tags, no revisions, no lock attributes
- the `.count` attribute gives the fixed size
- writing outside the size is a fatal error

They are implemented in the runtime system, but a few tests showed that the gain in speed is less than 10% in a program that heavily modifies rows of unchanged size (puzzle solve), and that they could not be used with many other examples, because the use of fields is very comfortable, so that lists could not replace rows in most cases. Other means, e.g. commenting debug prints and using the strongest optimisation, were more than 50% effective.

Thus, the added complexity (and additional chances for errors) are not considered worth the small gain in efficiency.

**Multi-dimensional rows**

As bunch values may be bunches, multi-dimensional rows can be created, but only by assigning rows to rows. Note that a row has to be created and does not come in to existence by assigning to a member:

```
a =: []
?# i =: from 1 to 11
    b =: []
    ?# j =: from 1 to 12
        b[j] =: i*j
    a[i] =: b
! a[3][4] = 12
```

This is really cumbersome, early compilers did not support two index expressions together; a variable had to be used.

Because the comma is already reserved for lists, i.e. the expression `r[1,2]` is a list of two row elements, a semicolon is used to index multidimensional arrays:

```
a =: [5;7]          \ create 5x7 matrix
a[5;7] =: 9
```

hence the attributes `a.first` and `a.last` deliver lists of integers, not integers. Internally, multidimensional arrays have to use one-dimensional memory, and the well-known formula to calculate the final index is used; thus, the expression `[1024;1024;1024]` allocates 1GiByte.

In any case, as with one-dimensional arrays, the sizes in each dimension are dynamically adjusted, implicating a lot of moving around the data to put it on the new place. So it is very strongly recommended to create the array with the proper bounds whenever possible.

## Template permutations

Instead of parameters by keyword, TUF-PL defines permutation of parts of the template, including optional parts, using the + and - characters. The + starts a template part that may be exchanged with other parts, and - starts an optional part.

The function with the template:

```
copy + from (a) + to (b):
```

may be called as:

```
copy from x to y
copy to y from x
```

Or:

```
from (start) + upto (end) - step (step):
```

may be called as

```
from 1 upto 7
from 1 step 1 upto 7
from 1 upto 7 step 1
```

Parameters in optional parts are set with void when calling, i.e. are just functions that call the full template with voids as required.

In order to avoid confusion, the + and - characters must be followed by a word, not a parameter, so that the word(s) behind it are the equivalence of keywords. Also, the partial templates must be unique.

This feature is not to be expected in early compilers.

## Threads

Threads — as implemented by most operation systems — split a process in two (or more) processes sharing all global data. Threads were very valuable in Microsoft's Windows systems, when neither UNIX fork nor shared memory was supported, and starting another independent process was slow.

As both processes execute instructions (not statements) in an unpredictable order (unless synchronisation primitives are provided and properly used), the execution of such a programme becomes unpredictable.

For example, the code snippet:

```
$glovar =: 1
a =: 1
! $glovar = 1
```

will often work, but sometimes stop because the assertion is violated. Of course, this assumes that `$glovar` is changed somewhere else, and this may well be done between setting and reading the variable. It will occur rather seldom, as threads normally run for many thousands of instructions before re-scheduled, but it will occur finally, depending on the load of other processes in the computer.

It is admitted that the same thing may happen when instead of the assignment, a function is called, that may or may not change the global variable, depending on the state of the data used, etc, and this is one reason that global variables are discouraged to use, unless good reason is given do do so.

All this is thoroughly discussed in Edward A. Lee's article *The Problem with Threads*, published in 2006, available online at https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html.

As strings and numbers are immutable, they do not conflict with tread safety, if the storage management is made thread safe as part of making the runtime thread safe.

Currently, access to the loop context uses a global variable which then must be replaced thread local storage (POSIX TLS). Another solution would be to add an extra invisible parameter to all function calls in the context of the right hand side of an assignment that is used in an enumeration loop, and defining that extra parameter if a function contains the loop context symbol. It would create a special class of scan functions.

Using only bunches as globals and the `.hold`, `.lock` and `.revs` attributes, a high degree of thread-safety can be achieved. Note that the `.scan` attribute uses the `.revs` attribute to protect against changes while a the elements are enumerated, which also helps against problems induced by threads.

As POSIX threads denote a function and a single parameter, starting a thread would be just a function (instead of a special digraph):

```
posix thread start `function using (parm)`
```

More functions are likely to be necessary in a separate POSIX thread support module.

**Alternate scan function schema**

Instead of using a global variable — which gives problems with threads --, functions that are scan functions can be provided with an extra parameter; the digraph `$#` could still be used for this.

The .....

To ensure that this extra parameter is propagated if there is an intermediate function, For internal functions, it is sufficient that the loop context symbol `$#` is used in the body of a function.

These functions have an additional parameter, that is represented by the loop context symbol `$#` and hidden from the user.

If after the loop starting symbol `?*` a boolean expression is found, it is just a repeat loop. Otherwise, there must be an assignment with at least one scan function. Then, a loop context variable is allocated, set to void and its address passed to the scan function. Upon loop termination — for whatever reason --, the loop context variable is dropped. There is no need to save and restore the loop context, as one for each nesting level is sufficient.

The formerly used scan loop symbol `?#` is no longer necessary (and in traditional syntax, `while` can be used for both).

While nearly almost the expression on the right hand side of the scan assignment is just the call of a scan function, any expression may be allowed, if there is exactly one scan function call in the expression. An example to count the slots modulo 10:

```
s =: [10]
?* i =: 1 + indices of row r %% 10
    s[i] =+ 1
```

No really useful example has been found so far, so the compiler might refuse expressions and insist on just one scan function call.

**Switches**

Basically, no case switches are defined. They were used in early times as an efficient multi-way branch, which is no longer necessary.

Admittedly, the statement example

```
? x = 1
    do something
|
    ? x = 2
        do otherthing
    |
        ...
```

is not exactly a terse expression; but the *else if* symbol (`|?`) makes the code fairly readable:

```
? x = 1
    do something
|? x = 2
    do otherthing
|? x = 3
```

```
    ...
|
    done otherwise
```

Using case switches for a large number of alternatives is not very often sensible, as the code is hard to verify, in particular if the cases are not just function calls. Using a row or array with function references is often a good and efficient alternative.

In many cases, the code might be organised without *else if*, e.g.:

```
? x = 1
    :> obtain first thing
? x = 2
    :> obtain second thing
...
```

which is fairly good readable, and is not restricted to equality comparisons.

The major disadvantage is that the variable `x` has to be repeatedly written, thus a shortcut to avoid this extends the `elseif` symbol:

```
? x = 1
    ...
|? = 2
    ...
|? > 3
    ...
|               \ the catchall (otherwise) case
    ...
```

The normal guard expression evaluation must be as simple comparison of a variable followed by a comparison operator, the former noted. When an *else if* operator `|?` is used and directly followed by a comparison operator, the noted variable is assumed to be prepended.

As the value of this is debatable, few compilers will support this extension.

### Macroprocessor

In the early days, a macroprocessor (or preprocessor) was useful for a slim compiler and as a mechanism to extend the language. As its purpose is to provide features not defined in the language, it spoils the structure and makes it harder to verify a programme. Thus, a build-in macroprocessor is not thought as appropriate for TUF-PL.

Just in case the demand for such a beast cannot be rejected, here is the preferred specification:

- A macro definition begins with a double backslash (\\) in column 1 and takes a full line
- Any sequence of tokens upto the mandatory second double backslash is the pattern.
- The remaining tokens of the line are the replacement
- A backslash character followed by a single plain letter is a placeholder for a token to be matched
- Any other token is matched verbatim, and white space is matched as white space
- Placeholder tokens in the replacement are substituted by the matched value
- Strings in the replacement are scanned for placeholders, which are substituted
- A placeholder not in the pattern is empty and not flagged as an error
- If there are duplicate placeholders in the pattern, the first one determines the value, and the following ones must match the value already set
- Matching and replacement is on a single (logical) line only and cannot change indentation
- A placeholder token cannot be matched, nor generated in the replacement.

All macro processing takes place once a line is parsed into tokens (words, numbers, strings, single characters and character pairs (digraphs); comment and pragma lines starting with a backslash in column 1 are not processed at all, as are trailing line comments.

Macro definitions are valid for the lines that follow, they can be redefined without error or warning; no second double backslash or nothing behind it erases the macro.

As an example, these would be the debug print macros (the question mark notation is not useable, as it

would match a guard):

```
\\ <\x> \\ '\x="' _ \x _ '"'
\\ <\a{\i}> \\ '\a{' _ \i _ '}="' _ \a{\i} _ '"'
\\ <\r[\i]> \\ '\r[' _ \i _ ']="' _ \r[\i] _ '"'.
```

Note that the increment assignment could be expressed as

```
\\ \x =+ \y \\ \x =: \x + \y
```

and the catenation with blank as a mere substitution by

```
\\ __ \\ _ ' ' _
```

Reserved words instead of symbols could be set as follows:

```
\\ if \\ ?
\\ else \\ |
\\ else if \\ |?
\\ loop \\ ?*
\\ scan \\ ?#
\\ return \\ :>
\\ repeat \\ ?^
\\ leave  \\ ?>
```

Of course, these words become reserved words then and cannot be used for variables etc. any longer, and it may give confusions as the compiler might show the generated line.

Note that macros do not match text inside strings of the source line, only in the replacement. Of course, a known string could be matched verbatim and replaced:

```
\\ "hello, world" \\ "Hallo, Welt!"
\\ 'Have a ' _ (u) _ ' day' \\ 'Wünsche einen ' _ (u) _ ' Tag'
```

Error messages refer to the replaced line.

Note that the backslash is used in the language for comments and pragmas only, so there is no need to have them in the pattern or generate them in the replacement.

### Coroutines

As an alternative to callbacks and threads, coroutines (see http://en.wikipedia.org/wiki/Coroutine) are supported in a generalised message passing model, that allows bidirectional value (message) passing.

Coroutine send and receive are denoted by port strings postfixed with the digraph <> resembling a special kind of array. It is in the send context if used on the right hand side of an assignment, and a receive context otherwise.

A function is suspended if the port in the receive context has no value, or for a port symbol in yield context the previous value has not yet been received. The compiler may thus use threads and parallel execution, but may also suspend before a receive port until the value is ready, or after a yield until the value is received, so that there are no threads necessary. (See ??? remarks).

Example:

```
Port = 'portName'
func1:
    ?# x =: Port<>          \ receive
        <Port> =: x*x       \ send
func2 (p):
    i =: 1
    ?* i < p
        <Port> =: i
        j =: <Port>
        print i '^2 = ' j
 func1 <Port> func2 10      \ start two coroutines
```

Function 1 starts and runs until the port is used in receive context; it is suspended until function 2 yields a value, which may be suspended or allowed to continue execution. Then, function 2 is suspended, the value transferred to function 1, and function 1 resumed until it yields a value, when it is suspended again and function 2 resumed, given the value supplied by function 1.

Coroutines are not available in all implementations. Implementing them with threads is critical, as unpredictable behaviour may occur if global variables are shared. A compiler may issue a warning, if global variables and coroutines are used in the same module.

### Unified Rows and Arrays

It would have been possible to syntactically remove the distinction between arrays and rows, and allow the index to be either a string or an integer, using the `bunch[idx]` syntax for both, rows and arrays, and creating both with the `[]` syntax. The item such created would be a neutral bunch, which may be used as a row or an array, depending on whether the index used is an integer or a string. (Although for an array only a comparison for equality is required, other indices than strings are rarely useful.)

If the first use would set the kind of the indices, and using the other one leading to a runtime error, the situation would be nearly the same as now, only that more runtime errors are expected because there is no syntactical distinction, and it would be more difficult to check this at compile time.

Allowing both, row and array access, to co-exist for the same instance, is possible, but has some consequences:

- there would be two attributes, `.ArrayCount` and `.RowCount`, unless the `.count` attribute would only report the number of array elements; and the number of row elements be calculated by `.last - .first + 1`.
- there would be two attributes `.ArrayEnum` and `.RowEnum`, unless only the array indices are given by `.scan`, and the row indices are by `from row.first upto row.last`, including those elements that have void references.
- the cells accessed by the integer `1` and the string `'1'` are different.

These are not really profound reasons, but all together let me prefer to keep the distinction.

A problem in the case of overriding the get and put functions was none. To support hash tables with strings as indices, it is tempting to have the primary bunch as a row, indexed by the hash values, each element of the row being a reference to an array, which is indexed by the strings supplied as arguments for the get and put functions. In this case, a row would be indexed by strings. This can easily solved by using an array as primary, with a field referring to the row of hash slots. As a consequence of this evaluation, overriding get and put is only valid for arrays.

The desire to use swivelled braces for the condensed notation is also solved, see ???Condensed.

### Shell usage

This is a rather old section that has to be thoroughly revised.

TUF-PL can be used instead of *bash* or *dash* etc, as its syntax is very terse. The major advantage is that you can use the same syntax in your shell as in your programming language.

However, there are some not so simple differences.

The `tufsh` shell can use any TUF-PL program as script, but, as it reads commands interactively, has loosened input syntax rules:

- The hash or number sign in column 1 indicates a comment line (in particular in line 1 to support the `#!/bin/tufsh` notation)
- Words that are not found are treated as strings
- Filename completion is enabled
- Matching uses Bourne Shell mode
- all programs in the path are functions, the file execution operator is not needed
- File name expansion, `%`, replaces the token by a list using Bourne Shell pattern matching

These relaxations apply only for terminal input, scripts have to adhere to the normal TUF-PL syntax.

In particular, file name expansion is done by the `files` function, which uses two versions, which use

Bourne shell patterns or regular expressions:

```
qls 'x*.jpg'
ls 'x*.jpg' \ because the sting is a RE, the point must be backslash escaped

%file (files '*.jpg')
```

All filenames have to be quoted:

```
rm 'x.out'
```

No automatic file name expansion, you write

```
rm (ls '~*')
```

But you will then more often use rows to check the filenames:

```
f = ls '~*'
print f     \ check output
rm f
```

Postprocessing `ls` output to print size and filename, split without parameters splits on white space:

```
fl = ls '-l' '*.#x'
! fl.?: x = split @; print x[5] " " x:9 .
```

This gives problems with filenames with blanks; better use the built in file functions:

```
fl = directory list '*.#x'
! fl:? x = @; print x.size, x.filename
```

To allow pipes, the symbol `->` is used in conjunction with the call operator:

```
%ls '*.jpg' -> %awk '{print NR, @0}'
```

Redirection is done by providing a filename as string as source or target:

```
ps 'faux' -> 'x.out'
'x.out' -> less
```

This does search for `x.out` in the path; this would be done by using the call operator explicitly:

```
%'x.out' -> less
```

To redirect file descriptor 2, i.e. standard error:

```
a.out -2> 'x.err' -> 'x.out'
```

To join `stdout` and `stderr`, just omit the filename:

```
a.out -2> -> less
a.out -> (1 2) less
a.out -> join 1 2 -> less
a.out -> ftee 2 'x.err' -> less
```

The equivalent to `cat`, i.e. write to a file, is named `tac` and both are provided as a built in functions, if desired for clarity:

```
f = 'x.out'
%ps faux -> tac f
cat f -> %mail me
```

If the string `x.out` should be send as mail, `echo` has to be used:

```
echo 'x.out' -> mail 'me'
```

Or to use my favourite mail test interactively:

```
date -> mail me
```

The special pipe function is activated if the source or target are executables in the path (selected by the percent sign unless in interactive mode). Otherwise, is is a common assignment:

```
5 -> x  ??????????????????
%mail 'me' <- %date
%mail @User <- echo 'Hi, how am I?'
```

Just for the interactive use, `echo` is also a build in function.

Instead of `%ls`, the build-in function `dir` can be used. It provides a row with information from the directory, as row attributes:

Instead of <code>%ls</code>, the build-in function <code>dir</code> can be used. It provides a row with information from the directory, as row attributes:

f.size

```
    file size in bytes
    file size in bytes
```

`f.name`

```
          file name
          file name
```

`f.permissions`

```
          file permissions
          file permissions
```

`f.inode`

```
          inode number, if supported by filesystem
          inode number, if supported by filesystem
```

`f.change`

```
          date of last change (seconds since epoch)
          date of last change (seconds since epoch)
```

`f.created`

```
          creation date
          creation date
```

`f.accessed`

```
          last access date
```

Note that `dir` does not filename expansion.

To rename several files, the following `bash` script might be used:

```
for x in x_*.jpg
do y=@{x/#x_/y_}
   mv "@x" "@y"
done
```

In TUF-PL, this reads, if typed in:

```
?* x =: %'x_*.jpg'
  y =: sub '^x_' by 'y_' in x
  %mv x y
```

Both can by typed in in one line:

```
for x in x_*.jpg; do y_={x/#x_/y_}; mv "_x" "_y"; done
?* x=: %'x_*.jpg'; y = sub '^x_' by 'y_' in x; %mv x y .
```

Note that filename expansion operator is used instead of the `ls`-command, because shell has to provide

filename expansion.

### Attribute and field alternatives

In the early design phases, in particular in search of a solution for the integer division that delivers two values, using the field notation for attributes was felt very elegant and logical, and is still considered handy.

But sharing the namespace of fields and attributes is debatable, because:

- Establishing new attributes may collide with fields already in use
- Misspelled attributes are treated as fields and thus may be unexpectedly void,
- Understanding a programme requires to know which tags are attributes
- Localisation should keep the English names and provide another set, increasing confusion.

The solution given above in Fields and Attributes is regarded a good one; here some remarks are collected on this subject.

The new rule that attributes to be changed must start with a capital letter concerns in particular the `.checked` attribute of fault items.

Practically, collisions of attributes with fields were seldom (in particular, as an expert was writing the few example programs), but in misspelled attributes occurred, in particular after renaming `.error` to `.isfault`.

If attributes cannot be changed, confusing them with field will be detected as soon as they are attempted to initialise. The remaining — writeable — attributes are `.tag`, `.checked`, `.byteorder`, `.lock` and `.hold`.

The attribute `.tag` behaves like a field until an already set value shall be changed, so there are no undetected mistakes.

The `.checked` attribute is used for fault items, not for bunches in general, and thus is now a field.

Remains the `.lock` attribute. Setting it with not an integer will be a runtime error. If set with an arbitrary integer, the inability to change the item is considered to be detected early.

Another alternative would be to start all attributes with a special character to form a digraph, e.g. `.$attr`, `.!attr`, `.^attr`, `.#attr`, `./attr`, `._attr`, `.&attr`, `.%attr`, `.,attr` etc. This is considered clumsy and not appreciated.

As a third alternative, a tilde, circumflex, exclamation mark or number sign could be used instead of the dot:

```
print x~count
print x#count
print x^count
print x!count
?# i =: row!scan
?# i =: row^scan
?# i =: row#scan
?# i =: row~scan
```

Using the exclamation mark or the number sign do not require changes elsewhere, as the exclamation mark is only used as the first character in an assertion or constraint, while the number sign is only used as second character in a digraph. Both are fairly well readable, although still a bit to heavy.

# 11. Implementation Remarks

The most direct approach that has found to work is to use memory addresses (pointers) for all kinds of items, having a uniform and robust, but possibly not fastest implementation. Alternatives are discussed at the end of this section.

## Storage Management

In order to automatically free storage that is no longer used, all such items have a use counter, which, when decremented to zero, indicates that the item is no longer used and can be recycled. When a variable gets out of scope, its use count also is decremented, as to free the space of the item.

For keeping track of the use counts, two macros are defined:

```
#define _use_less(x) if (x) { --(x->usecount); if (x->usecount <=0) _free_item(x); }
#define _use_more(x) {if (x) ++(x->usecount);}
```

Thus, if a reference is copied from one variable to another, the target reference is referred to one less, and thus the use counter decremented. The use counter of the source reference, as the latter is copied, must be incremented.

A straightforward macro does this:

```
#define _cpy_var(tgt, src) { _use_more(src); _use_less(tgt); tgt = src; }
#define _cpy_ref(tgt, src) { _tmp = src; _use_more(_tmp); _use_less(tgt); tgt = _tmp;
```

However, the macro `_cpy_var` can only be used if `src` and `tgt` are simple variables; if `src` would be a function call, it would be evaluated twice. Also, it may use the contents of `tgt`, thus `tgt` must remain valid until `src` is finally evaluated. Note that for the case `src=tgt`, first `src` is incremented, `tgt` decremented, also to avoid the intermediate deletion, i.e. to try to increment the use count of a no longer existing item. For the general case, the second form `_cpy_ref` must be used.

Note that if a local variable is used as a function argument, the use count is not incremented by the caller, different to the treatment of return values. As during the execution of the called function, the local variable of cannot be changed, there seems no need to increment the use count. If the called functions copies the reference to variable local to it, the use count will be incremented, but decremented if the variable gets out of scope. However, if the called function is allowed to modify the parameters, i.e. treat them as predefined local variables, the parameter's use count has to be incremented, as otherwise there would be two variables referring to an item with a usecount one to less.

As this argument is not valid for global variables, their usecount must be incremented and decremented properly. If If the called function replaces the reference of that global variable, if will not be freed until the function returns, and thus its value remain valid.

The return values of functions are in a certain sense transient: They are no longer references contained in variables, thus the count of references from variables might be zero, but they cannot be freed until they are finally disposed of.

Let us assume a function just created a new item and returns its reference. This item must still have a use count of 1, as otherwise it would have been deleted. In case it is stored in a variable, this can be done by decrementing the usecount of the previous contents, and then just saving the reference in that variable, without incrementing the use count of the function value. A macro is defined for this, under the assumption that `src` is not an expression that might use `tgt`; otherwise use the second form:

```
#define _set_var(tgt, src) { _use_less(tgt); tgt = src; }
#define _set_ref(tgt, src) { _tmp = tgt; tgt = src; use_less(_tmp); }
```

Note that `_cpy_var` could be replaced by two lines, but this does not save any effort in code generation.

```
_use_more(src);
_set_var(tgt, src);
```

Alternatively, the transient values could have a use count of zero, and the above macro `_cpy_var` be used to set a variable, which would make code generation easier. See below for more on zero use count transient items.

More difficult is the handling of nested function calls, where function results are used as parameters of other function calls. This is tempting to generate, but the problem arises how to free the transient return values.

A simple and robust solution would be not to used nested function calls, but assign all intermediate values to temporary variables, and when the statement has finished, release all these temporary variables. Still, the use of `_set_var` instead of `_cpy_var` would be required, but the information

whether the value is a variable or an expression is present in the compiler.

Note also that functions may be called discarding the return value; in this case, the returned transient value must be disposed of, either by using `_use_less` in case of return values with nonzero use counts, or a `_drop` macro:

```
#define _drop(x) { if (x) { if (x->usecount = 0) _free_item(x); } }
```

Using transient items with a use count of zero would allow direct function nesting and uniform handling independent of the source of the value, if extra effort for a kind of garbage collection for these transient values at the right places, which both is not simple, would be invested.

First, reducing the use count would require a different action, as it is not free when he use count is zero, but possibly entered in lists for transient values.

If all items are anyhow kept in a list, this list could be scanned for zero use count entries, and these deleted. However, not chaining the used items saves memory and processing time. Alternatively, only the zero use could be kept in a list.

However, there is no point in time where all these transient values may be freed. Let us just assume that the main program contains the line

```
x =: fun1(fun2 1, fun3 2)
```

This means, that until the return of fun1, the transient return values of `fun2` and `fun3` have to be kept. One solution would be to set aside the current chain before the call of a function, after return clear this chain, and restore the chain. It is not clear if this elaborate system will give a benefit at all in run time and space over the simple solution of just to avoid direct function calls.

Note that the advantage not to handle each return value individually is bought by handling it before return, and additional effort for garbage collection.

In general, using reference counts, variables are concerned. If a variable is assigned a new value, the old reference is decremented (and the object freed if zero), and the new reference incremented. So this is a reference copy, implemented by the macro `_IPcpy(tgt,src)`. If the (old or new) reference is void, nothing has to be done. At the end of the block, all variables are decremented, so that, if they have the last reference, the item is freed.

A variable that is parameter fits into this scheme; it remains existent until the function returns and the variable is overwritten. Parameters that are constants could be marked as such, and never need to be freed; in particular, if the same constant has to be itemized again, the same reference could be used.

If a function returns a reference to an item, it must be preserved until it is assigned to a variable, it becomes transient. Thus, its usecount is not decremented; otherwise it will be deleted before it can be used. When assigned to a variable, the old reference is clearly to be decremented, but not the transient reference, as it it just to change from a transient to a permanent state once stored in a variable location. Thus, a different macro, `_IPset(tgt, src)` is used.

If a transient value is not made permanent via a variable, the caller must ensure its usecount is decremented once it is no longer needed, in particular, if it is discarded. So, if a function is called without assignment, its return value has, if not void, its reference count immediately decremented, and in many cases just then deleted.

This prevents to compile nested function calls directly, as the return value must be discarded when the outer function returns, and the intermediate value have to be saved in temporary variables for this purpose.

Independent of the usecount, the value returned by a function must be properly administrated by the caller of the function. If the value is assigned to a variable, and it has already the final usecount, n

The problem comes with expressions where the intermediate values, mostly return values from functions or operators, must be kept until the functions or operator for witch they are operators returns. Thus, the simple solution is to use temporary values, assign to them the return values, and destroy them once the values are used.

The current version of the macros reflect these problems. Note that up comes before down, in case the target already contains the same reference as the source, because otherwise the source may be

deleted in between.

A fairly tricky solution might be to queue all values returned in a transient queue, which is cleared after an expression (that has created transient values) is done. However, as another function can have already created transients, there must be mark in the transient chain, so that the solution to used temporaries seems more efficient.

The run time might use indirect references and, on a 32-bit system, use 24 bits for integers, and use the excess numbers as references. Thus, instead of items, only integers are passed to functions, where a global array maps integers to items. Integer 0 is reserved for void.

## Rows and Arrays

Rows are kept as continuous memory of item pointer, re-allocated if necessary. Currently, the amount added is fixed to 10, which seems rather suboptimal for large arrays. But no real performance issues have been observed: Calculating 10 million primes, i.e. increasing the array a million times, in fact has a short delay while the multiples of the small primes are written into the table, but the difference between the first and the second prime is not much larger than expected.

Arrays as well as user-attributes use a unidirectional linked list starting with the newest entry. This is clearly a performance bottleneck if a large number (>1000 or >10000?) of indices is used, in particular during insertion of new keys, as this requires to search the whole chain.

To speed up array access, a hash table could be used, either by implementing it in TUF-PL, or by integrating it into the runtime system.

The default hash function is the length of the string, so it remains the number of slots to be set. This can be done in the same way as the estimated size of an array is given, i.e. as a number within the bunch creator. If consistency with the row creator is most important, it is the estimated number of elements. However, using the stringlength as hash function, 10 slots is almost optimal choice. Using a TUF-PL solution, the speed is increased by approx. the factor 5. Thus, the rule remains that the number is the estimated number of elements, but the standard implementation will simply switch to 10 slots if a number is given.

## Tagged bunches

It might be an implementation option if bunch functions are connected to a tagged bunch during creation, so that the runtime code can search the bunch. However, the current solution seems to be sufficient.

## Level 0 TUF

For bootstrapping, a restricted version of TUF, *level 0 TUF*, is used:

- no list assignments
- RHS of an assignment has only one operator (no real expressions)
- no floating point (only integer, C-strings and bunches, the latter however, with all features.)
- no top-level statements, use `main(params)` function
- no boolean expressions, only comparisons.

### Strings

In order to avoid copying of substrings, the string could be a row of substrings, where each element holds the reference to the source string and the address plus length pair of the substring. A method can then collapse the stuff to a single new string, which will also be required if substrings of substrings are required.

The idea came up considering a text file to be opened in shared memory; then, the lines are rows of substring pointers, just excluding the newline characters. For this purpose, a string should be able to specify the memory pool to which the memory should be returned:

- none, i.e. from program space or embedded
- malloc, i.e. directly allocated
- string item, i.e. have usecount, and finally one of the other

### Named Constants

Many libraries have constants as parameters, that are (in C) defined as macros, i.e. `FL_PUSH_BUTTON`, providing short integers. During early times of computing, the efficiency gain might have been significant in some cases; this is truly no longer the case nowadays.

TUFPL prefers (short) strings, so that instead of:

```
    xforms add box FL_UP_BOX title "Quit?"
it is preferred to use:
    xforms add box 'up' title "Quit?"
or define a named constant:
    #FL_UP_BOX='up'
```

Constraints can be used to restrict the strings at compile time:

```
    ! FL_UP : 'box' | 'nix' | 'nax'
    xforms add box (FL_UP: flag) title (str:string):
```

In the basic implementation, short strings do not take up more space than integers, and comparison is quick compared to the overhead required anyhow.

### Library bindings

Binding foreign libraries is usually done by providing an interface module, that provides the features of the foreign library by TUF functions. These may, but often will not be a 1:1 relation in order to better adopt the interface to the programming concepts of TUF-PL.

Practically always these binding interfaces need to use data structures different from the items used in TUF-PL, i.e. opaque memory.

For this purpose, there is a system item subkind for foreign (opaque) data. The library may well return bunches which contain normal TUF items, like numbers, strings, etc., and use one or more fields to store foreign data.

The payload for a foreign system item contains two pointers:

- an opaque data pointer
- pointer to a destructor function

If the destructor pointer is not zero, it is used to call a function using this pointer, with just the data pointer as argument, before the item is finally destroyed. The function should return zero; otherwise the situation is treated as a fatal system error and the programme is aborted. There is no means to avoid having the item memory freed afterwards. Note the argument is not the item, just the data pointer.

It is strongly recommended to start the opaque data with a magic number, so that any library routine can check — with high probability — that the pointer in the item provided had been created within this library interface and could reliably be used as a trusted data structure. The preferred magic number is a pointer to a string literal within the module, which is unique within the module. The string may be `org.tufpl.bindings.xxxxxx` where xxxxxx identifies the interface.

---

[1]as was shown 1970 by W. M. Waite in STAGE2 http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.98.965&rep=rep1&type=pdf.
[2]which is a wrong term anyhow, as *object oriented* is a design method, and can be used in any programming language, more or less easy
[3]I would very much appreciate if there are volunteers to try more examples.
[4]meaning not equal, but with small differences that often break the program.
[5]That's why the ALGOL assignment digraph `:=` is not used.
[6]The context variable is just a hidden global variable, if no threads are used. Otherwise, it must be ensured that the context variable is not shared by threads. Because it is global, the digraph is `$#`.
[7]In early days, the programmer war trained to do arithmetic by bit operations which is no longer needed as most compilers can do a better optimisation.

[8]Although it may be that the bunch is locked, in order to avoid accidental overwriting of vital information.

[9]The current runtime system has currently implemented only 32-bit rationals for a prove of concept.

[10]In this case, the runtime must check for overflows and terminate the programme once an overflow is detected.

[11]Raymond T. Boute: *The Euclidean Definition of the Functions div and mod*. ACM TOPLAS vol.14 no.2 April 1992, pp. 127-144.

[12]Also among the conditions is `n÷d ∈ Z`, because he extends the criteria to real numbers. The condition `n†d ∈ Z` had probably been left out to allow real numbers for the remainder.

[13]Coincidently, in the current implementation, an array without elements needs less memory than a row without elements.

[14]in contrast to AWK, where using the index for an array creates the cell

[15]Flow analysis could nevertheless give a compile time warning

[16]Actually, the last condition should read `(100*@).den = 1` or `@.den = ( 100, 50, 25, 10, 5, 2, 1)` if the latter is supported by the compiler

[17]The first runtime uses such a construct to save space for short strings not exceeding 6 characters.

[18]like the lifeboats are removed once a ship is used daily

[19]To use an exclamation mark to bind a constraint had been considered, but rejected as more clumsy than using consistently the double colon.

[20]An empty string is not void, thus it is stored in the row as any other value.

[21]If the attributes are a chained list of name-value pairs, use a second list for name-function ref pairs.

[22]except that is is not possible to extend the language later to allow =< for *equal or less*