The TAV Programming Language

Rainer Glaschick, Paderborn
2024-11-04

This is version 0.10.1 of TAV-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 some 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. Also, it encourages very complex expressions that are hard to understand at the first glance — from my experience, easy understanding of programm lines is a prerequisite for reliable programming.

As was shown 1970 by W. M. Waite in STAGE2[1], it is possible to intermix parameters and words in calling a function, avoiding the hard to read parenthesis mountains.

The most prominent features are:

- Terse without keywords in the core, just symbols
- Yet verbose, as functions use many words and have parameters intermixed
- No type declarations required for variables
- Parenthesis are used sparingly due to indentation awareness and function templates.
- Compiled to C, thus efficient and easily provided.
- Arrays and maps grow dynamically.
- Memory management by use counts avoids unsolicited garbage collection delays.
- Frugal without introspection and self-modification

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. (with a grain of salt concerning attributes.

The language is imperative, with a support for object oriented programming. Note that *object oriented* is a design method, that can be used in any programming language, admittedly more or less easily in one or the other.

Any attempts to save typing effort by using e.g. int instead of integer were discarded, as clarity is the main goal, no saving a bit of time while typing, while spending time later to debug and modify the code.

There is no compile-time enforced strict type system; however, the constraint mechanism — if supported by a future compiler — is deemed to be 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; It can be fully localised, as there are no reserved words, like if, else, function and the like. However, the standard library uses English terms, and attributes are also close to this common language. Nevertheless, UTF is fully supported; although not yet used in function templates. For European users, only basic ASCII is used.

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 a 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 maps with fields can be modified affecting the caller's view, contradicting the functional programming view.

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, some tests with Python and AWK showed that equivalent TAV-PL programs are often quicker.

Graphical programming libraries have been applied, but only of a very rudimentary level of *proof of concept*. This is defintely a huge task for the furture.

In order to publish code (and perhaps to attract programmers used to keywords), instead of bigraphs starting a statement, e.g. ?, word tokens, e.g. if, like in nearly all other programming languages may be used if enabled by a pragma.

The name is *TAV* and *TAV-PL* for historical reasons[2], but no longer an acronym. In order not to preempt tuf-pl.org for Polish users, the domains used are tufpl.de and tufpl.org, both without hyphen. *TYV* for *Terse Yet Verbose* would be more appropriate, but means *thief* in Danish; so what is a name?

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 are for example:

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, hard to understand and hard to verify.

Lines

As of the genral rule, a statement is contained in one line.

It might by possbile to automatically detect line continuations, if a line ends in an operator symbol, or parenthesis are still unbalanced. However, the benefits are small compared to the larger obscurity.

Consider to break down long statements using extra variables; this will neither increase execution nor compilation time, but increase readability and thus reliability. Breaking down a complex expression allows to comment the parts individually.

If it cannot be avoided, or just for better layout, a join of two lines can be forced by placing three dots in a row (...) as the last symbol of a line (white space including comments may follow and is ignored). The compiler appends the next line to the current one replacing the triple dot upto the end of the line by a blank space, and clears the next line. If the new combined line again ends with a triple dot, the process is repeated. Leading whitespace of the line to be joined is removed, thus its indentation is immaterial.

Because the line is joined to one long logical line, error messages will show the tokens of the joined line and give the number of the first line.

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
- void statements

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

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

White space must be used if necessary to separate tokens, but is optionally otherwise. A symbol may be composed of one or two characters (neither letter nor digit) not separated by white space, i.e. 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, and that it may no start or end with an underscore.

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

- An open parenthesis starts recursively a subline, e.g. 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
- a string constant
- a word, i.e. a letter followed by letters or digits
- an operator symbol (unary or binary, including the double backtick)
- a sublist

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

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.

As empty lines are ignored and a comment is treated as white space, an explicit empty (do nothing) statement is denoted by just the void symbol `()`, just instead of a loop repeat (`?^`).

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 special 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

Three types of comments are possible:

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

Line tail comments:
: After a single backslash `\` followed by a blank the rest of the line is a comment.

Conditional comments:
: The digraphs `\?` and `\~` allow comments conditionally.

Note that backslashes have no special meaning in string literals, so comments start and end always outside of string literals.

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.

Despite the use of parenthesis, nesting comments is not (yet) supported; the result is undefined. Instead, the digraphs `\[` and `\]` are additional pairs for block comments, that are intended to exclude parts containing block comments unconditionally.

A simple mechanism for conditional compilation uses the enable command line option `-e` of the compiler and the digraphs `\~`, `\{` and `\}`:

- Normally, i.e. without enabling via `-e`, `\~` starts a line tail comment, and `\{` a block comment until `\}`.
- When `-e` is given, the digraphs `\~`, `\{` and `\}` are ignored.

If enable is set, the digraphs determine the indentation level:

```
do something:
   i =: 1
   \~ i =+ 1
   \{ i =+ 3 \}
   print i     \ prints 1 normally, and 5 if -e is set
```

This allows to include test code in a module:

```
a func to export:+
   ....
an internal func:
   ....
\{
main (parms):-
   a func to export
   an internal func
\}
```

More flexible conditional comment blocks use the digraph\? in column 1, followed by a condition, which may be a logical expression using only literals and named literals:

```
#MAIN = ?+
\? #MAIN
main (args):+
   ...
\?
```

If the condition is true, normal processing continues; otherwise — if false — the following lines are treated as comments upto and including a line that starts with \? and defines the next conditional comment block.

For convenience, any result that is not void, zero or boolean false is treated as true. Undefined named literals provide void and thus start skipping lines; no ifdef is necessary.

No condition at all is a true condition, although elsewhere void is treated as false; thus, a line just starting with \? and empty otherwise ends the conditional comment block. The same could be achieved by \? 1 or \? ?+.

A conditional comment line may contain a line comment, i.e. any backslash on the rest of the line ignores it and the rest of the line. Block comments are not recognised, neither opening nor closing.

Setting a named literal may be done by a compiler option as usual.

Alternatives (*else*) could invert the condition:

```
\? ~#MAIN
```

or use \| instead, if available.

Early compilers may lack this type of comments totally or may only support a single named literal as condition, which must exist to enable the block.

Programmers might consider adhering to the following conventions:

- \- to denote disabled statements (alternative to\~)
- \T or \TODO to denote places to check
- \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 comment blocks.

Some other digraphs are already defined (see next section), but may not be honored by the compiler 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 and identifiers are broken by comments. Backslashes in string literals are no comments; the are copied verbatim (including the following characters).

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

Pragmas

For modularisation, there are 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, named literals and constraints marked correspondingly. (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. (For this to work, the header extraction must be idempotent.)

If the name is not a pathname (starting with a slash or a dot), 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, TAVINC environment variable) until the file is found. If not path and not found, TAVPL_ is prepended and .tufh appended, and the search done again, so that the standard include might be

```
\+ stdlib
```

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, as its use and consequences have not yet been analysed.

The pragma \A enables (and \a disables) the automatic conversion of accurate (integer and rational) to rounded (floating point) numbers for the usual arithmetic operators because of the potential loss of accuracy. With \a, the accuracy of arithmetic expression is not compromised and must be stated explicitely (by using the .float attribute or corresponding function). A conversion back from floating point numbers to accurate numbers is never done automatically.

Not yet implemented; since 0.10-3 is enabled per default.

The pragma \C enables (and \c disables) C-style syntax and behaviour. It allows the following keywords (and inhibits their use as variable) in addition to the corresponding symbols:

```
?  if
|  else
|? elif
?* while
?# scan    for ?
?^ repeat
?> break
:> return
!  assert
```

For clarity, the keywords cannot be used as variables names.

Additionally, a single = (in addition to =:) can be used for assignments. Also, a double equals (==) and its negation (!=) may be used as comparison operators in logical expressions in additon to the single equals sign (=) and the ~= bigraph. The operators & and | are still logical operators that require boolean values on both sides, and do not become bitwise and or or in C mode, because there are no such operators in TAV-PL (functions must be used).

Also, the assign-and-operate symbols +=, -= and *= may be used in C mode assignments, the modify operators=+, =- and =* are disabled in C mode, because a C programmer might write:

```
cnt=-1        \ must act as 'cnt = -1', not as 'cnt =- 1'
```

The auto-increment and -decrement operators are not supported at all, because they have no counterpart in TAV-PL. They are leftovers from former times, when it was sensible to support machine instructions in programming languages.

Example:

```
\C
herons root (n):
   x = n
   y = x - 1
   while y < x
```

```
        x = y
        assert x == y
        y = (x + n // x) // 2
    return x
\c
```

In C mode, the above keywords may not be used as variables. Using them in function templates is possible, but might lead to conflicts.

It is recommended to switch to and from alternate syntax only between functions, as only the function body is interpreted differently.

For nationalization, it is possible to map attribute words by the\. pragma, that sets the new word instead of the old one, e.g. for a German variant:

```
\.Ende   = Last
\.Anfang = First
\.Anzahl = Count
\.Art    = Kind
\.Klasse = Class
\.Marke  = Tag
\.Schloss = Lock
```

If the second word is one of the attributes known to the compiler, the first word is an alias for the second one.

The pragma \# may be used to ensure to indicate the language version required, so that defective code is not accidently produced, e.g. when new attributes are provided.

TAV-PL follows the rules of *Semantic Versioning* (https://semver.org/). Thus, language versions comprise of a major (version) number, a minor (release) number, and a modification (patch) level, e.g. 1.12.2.

Also, a number of keywords may be given 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:

```
Tuples
ClassFunctions
FunctionPrototypes
CallbackFunctions
FieldChangeCallbacks
```

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 check issues a warning if:

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

Also, the compiler could flag language changes, e.g. new or deprecated attributes, once the compiler (minor) version is larger than the one given in the pragma.

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*[3] 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.

Note that the double backslash starting a line is reserved for a macro feature.

Inline C statements

As TAV-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 TAV-PL. The canonical way to do so is to provide a short module in C using the TAV-PL interface.

It is very unlikely that inline C statements will ever be defined and implemente.

Deprecated Functions

If (exported) funtions in libraries are deprecated, it might be desired give a compiler warning when used.

For this purpose, the :~ digraph can be used instead of :- in definition files.

A compiler option (-x) allows to exclude all deprecated function declarations.

Currently, deprecated functions are marked in the source files by comment blocks starting with(~.

Using :~ instead of :+ is currently not considered as :- and :~ are declarations, and : and :+ are followed by a definition block. so maybe :? will be used.

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 from the string written, 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 items that can be changed are maps and rows (and a few more), which has also features of a structure. To create a new row or map, use the factory functions new row or new row. 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 map without elements, and"" or '' are empty strings, all not the void reference.

The digraph () might be misinterpreted as an empty tuple; however, there are no empty tuples, although(().) is the same as (),, a tuple with only one element, which is a void reference. Note the trailing comma, as tuples with one element are normally useless.

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 (quotient of two natural) 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 (or $(2/3)_{10}6$).

Floating point (*real*) numbers require a point between two digits, as in1.1; the short forms 1. pr .1 are not supported. As with the exact numbers, additional operators may be used, e.g. in 12.45*10^-6. The commonly used

form 12.45E-6 (or 12.45e-6 is supported due to its compact form, but a decimal point is required (the short form 3E5 is not possible). Using the subscript 10 ($_{10}$, the unicode character ⏨) is planned to be supported instead of E or e, e.g. 12.45_{10}-6, but not yet implemented.

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 from the programme text 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 the same in another locale.

In order to have large numbers better readable, in particular in assertions, groups of digits may be separated by an underscore (_) or an apostrophe (single quote,'), without spaces around the separator:

```
n =: 100'000'000
m =: 123_456_789
```

The underscore is accepted for compatibility with other programming languages, but beware of old compilers that treat it as string catenation. The apostrophe is regarded better readable, although its use as a string delimiter might be confusing.

Neither blank space nor a comma or point may be used in programme texts, even with localisation.

Numbers in other bases than 10 must in general be created from string literals using library functions:

```
o123 =: string '123' as integer base 8
x1AF =: string '1af' as integer base 16
b101 =: string '101' as integer base 2
```

For convenience, octal and hex (base 16) numbers may be denoted using the Python3 notation, i.e. the digit 0, followed by the letter o , x or b (also as capital letters):

```
o123 =: 0o123
x1AF =: 0x1AF
b101 =: 0b101
```

In contrast to C, leading zeroes do not constitute octal numbers, except when a single zero digit is followed by the letter o.

There is a library routine to convert strings according to these rules to an interger:

```
nn =: string (x) as integer literal
```

Note that because integer numbers are arbitrary large, this notation cannot create negative numbers:

```
! 0xffff'ffff > 0
! 0xffff'ffff'ffff'ffff > 0
```

If rows of bits (binary words) are required, use the bits library functions.

String Literals

String literals are one or more 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.

With the only exception of a newline character, any character that is not the string delimiter may be part of the string, including tab characters. This means that the closing delimiter must occur before the end of the line, except in multiline string literals (see next), avoiding cryptic errors lines apart if the closing delimiter is missing. In particular, either form 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="'
```

Additionally, the delimiter may be written doubled inside, as two strings of the same delimier must be separted by blank space:

```
print "<a href="""
"He said:""hu!"""    \ He said: "hu!"
'Hab" acht'          \ Hab' acht
```

Also, a HTML entity may be used:

```
print '<a href=&apos;'   \ prints: <a href='
```

So to encode special characters, HTML entities can be used (see https://www.w3schools.com/html/html_entities.asp).

If the HTML entity invalid, in particular the final semicolon is missing, no substitution is done; thus e.g.&sel. may be used as placeholder.

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

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

Examples:

```
"He said: &quot;hu!&quot;"  \ He said: "hu!"
'Hab&apos; acht'           \ Hab' acht
```

The HTML standard allows any code point using their numerical value, but was not intended to use code points as control characters, so does not have names for these. Pragmatically, TAV-PL uses names for these:

```
nl    &#10;     newline
cr    &#13;     carriage return
tab   &#9;      tabulator
sp    &#32;     blank space
bsl   &#92;     backslash
e10   &#9182;   subscript 10 (₁₀) for number literals
```

Example:

```
'One&tab;tab'
'line&nl;change'        :::
```

The encoding by HTML entities can be used without restrictions to create a string literal from any string using & for any amperands contained in the string, representing control characters by the hexadecimal notation.

In particular, the string delimiters as well as the newline and tab characters are quite clumsy to read.

As no two string literals follow directly without a space, the entities" or ' can be represented by doubling the delimiter, so that the string is termiated by an odd number of delimiters followed by white space (or any other non-delimiter):

The backslash (\) has no special meaning inside a string and need not be escaped; if desired so, it can be written as &bsl;.

Strings in general support zero bytes (zero code points), but not in string literals, as these are internally still zero terminated strings. A zero byte can be read from a file or created by using the code point 0.

Multiline strings can — an exception from the rule that the line is the unit — span several lines.

To start a verbatim multiline string, use <' while <" starts a localized variant; both are terminated by a single apostrophe or quote and may not contain the other one as for any other string. If they occur on the same line, they are treated as an ordinary string. i.e. the multiline string still cannot contain the delimiter and may be followed by more tokens.

The information between the opening and closing string delimiter is copied verbatim, i.e. all characters that are in the source code including trailing blanks and tabulator characters. (Compiler input is not automatically un-tabbed, only the indentation counting does so for the initial whitespace.)

Example:

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

Line changes are represented as newline (&nl;) character:

```
s =: <'
```

```
    a 1
    b 2
    c 3
    '
! s[1] = '&nl;' \ if there are no blanks at the end of the line
```

Any removal of unwanted white space must be done with one of the several string functions, as there is no universal strategy except copy verbatim and leave it to the programmer.

Nothing (except whitespace and a line comment) may follow the closing delimiter, so the multiline string may only be the last one for a function call. It is clearer anyhow to assign the string to a variable and use it.

A typical use is:

```
inp =: <'
    1   void
    2   integer
    5   real
    '
inp =: inp::without leading whitespace
inp =: inp::without trailing whitespace
?# lne =: string inp give parts by any of '&nl;'
    print lne
```

Another example is the *usage* message:

```
print usage for (progname):
    msg =: <"
        Usage: &prog [options] command [values] '
        Options may be:
            -x  execute
            -v  print version and exit
        THis is version &version.
        "
    print string msg replace many '&prog'->progname, '&version' -> 1.2
\ or
    msg =: <"
        Usage: %; [options] command [values] '
        Options may be:
            -x  execute
            -v  print version and exit
        THis is version %;.
        "
    print msg % progname, 1.2
\ or using percent formatting
    error print <"
        Usage: %; [options] command [values] '
        Options may be:
            -x  execute
            -v  print version and exit
        THis is version %f.1;.
        " % progname, 1.2
```

Note that placeholders for dynamic contents must be replaced as shown, while localisation is done automatically. Beware of leading tabs, that most often will be expanded by 8 when printed. Consider untabbing first.

String Localization

String literals in quotes (") are localized when used, i.e. each time when assigned to a variable or used in an expression (including parameters of functions).

This is done by the system library function:

```
string: string <@> localize:
    ? is string @ localized         \ do not apply again
        :> @
    lf =: $SYSTEM.l10nfunc          \ function instead of map?
    ? lf ~= ()                      \ yes
        rv =: lf:(@)
        ? rv != @                   \ if changed, set flag
            rv =: system set string rv lsl10n
        :> rv
    lm =: $SYSTEM.l10nmap
    ? lm ~= ()                      \ use map if defined
        rv =: lm{@}
```

```
    ? rv ~= ()
      \ will copy the string, as it comes from the map
      rv =: system set string rv lsl10n
      :> rv
:> @
```

A string in quotes that has not been localized, i.e. changed by the above function, but returned unchanged, has the attribute .DoL10n set to true (?+).

If it has been localized, the substituted string has the attribute.IsL10n set, but no longer .DoL10n. In this case, the string does not undergo localization again. It also available via the function

```
is string (x) localized
```

The above function system set attribute () lsl10n is not for general use. (Unless the reference count of the argument is 1, the string is copied and the copy has the attribute set.)

The two fields of $SYSTEM can be read and set directly via field syntax, while the flag setter must be a function as strings are immutable, and a copy must be provided (unless it's the only use of the string):

```
system set string () IsL10n
```

Tuple, row and map literals

Tuples, lists and list assignments are immutable lists of items. While tuples can be created by converting a row, very often a list of items is used. If this list contains only literals and tuples from such, it is a tuple literal:

```
t =: 1, 2, 3, 5, 7, 11              \ a tuple of six integers
t =: ( 'a', 1 ), ( 'b', 2), ('c', 3)   \ a tuple of tuples
t =: 'a'-> 1,  'b' -> 2, 'c'-> 3       \ same in pair notation
```

The bigraph -> is a binary pair generation operator of higher priority than the comma:

```
a =: 'a' -> 1, 'b' -> 2 \ tuple with one pair
! a.Count = 2          \ tuple with a pair as single element
a =: 'a', 1 , 'b', 2
! a.Count = 4
```

Because it is a binary operator, it is cannot be used to create tuples with more than two elements:

```
x =: 1, 2, 3
! x.Count = 3
y =: 1 -> 2 -> 3       \ is (1 -> 2) -> 3
! y.Count = 2 & y.1.Count = 2 & y.2.Count = ()  \ .Count of an integer is void
! x ~= y
```

A tuple can often be used where a row is only read, and can be converted to row by a standard function to have a row initialized:

```
tuple (@) as row:
```

Thus, a row can be created by

```
r =: tuple 1, 3, 5, 7, 11 as row
```

This can be abbreviated by enclosing the tuple in square brackets:

```
r =: [ 1, 3, 5, 7, 11 ]
```

From the practical point of view, a tuple is is a row literal, that will finally be available as named literal, provided that no variables are used:

```
#three = 1, 3, 5
```

Note that in the in the following assignment:

```
y =: r[1,3]
```

the part [1,3] is not converted to a row, as the bracket after a variable designates an index.

Can be used with the matrix library.

To force a list, a leading comma can be used, which is ignored otherwise:

```
x =: , 5
! x
x =: , 5->6

range (list):
   r =: []
   for i tuple list give values
      r[] =: i
   :> row r as tuple

operator ..?
```

Named Literals

It is sometimes useful to give numbers or strings a name to allow consistent use. This is done by named literals, that are words prefixed with the number sign or hash (#).

Setting named literals starts with a number sign (#) in column 1, followed by a word, followed by an equals sign, and a single value (to be extended to expressions):

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

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

Setting a named literal may be only done outside function bodies, i.e. the defining line must not be followed by an indented block.

The value maybe any literal or named literal defined lexically before. Compilers should support expressions of literals and named literals. Despite the use in the definiton of named literals, they can be located anywhere in the source code. It is recommended to define them lexically before the line of use too.

Named literals should not be redefined. As the compiler might allow to set named literals in the command line, and to avoid libraries to disable words, a second form allows redefinition by using the assignment digraph:

```
#a = 1
#a =: 1
```

Note that the lexically last one wins for replacement in the total programme text, while the actual one is used in conditional comments.

To use a named literal, the same word with the number sign prefixed is used. While this looks clumsy first, it is absolutely clear and avoids rules about when a literal and a variable is meant; so it is consistent with globals.

To indicate that a named literal is exported, i.e. copied by the header extraction and the \^ pragma, it starts with a double number sign ## in column one. Within function bodies, it is still the single number sign with a word following, so that the double number sign for converstion of accurate number to rounded (float) numbers is still available (even if its use is discouraged). Using the dollar sign instead of the number sign was discarded from the same reason.

Some number of common constants are defined in the standard library:

```
\( Numerical constants
\)
##MaxInt31 =           2'147'483'647
##MaxInt32 =           4'294'967'295
##MaxInt63 =   9'223'372'036'854'775'807
##MaxInt64 =  18'446'744'073'709'551'615
##Pi = 3.141'592'653'589'793'238'46
##E =  2.718'281'828'459'045'235'36

\(  String constants
\)
##Whitespace = ' &tab;&nl;&cr;'
##ASCIIupperLetters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
##ASCIIlowerLetters = 'abcdefghijklmnopqrstavwxyz'
##ASCIIletters =  'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstavwxyz'
##DecimalDigits = '0123456789'
##HexDigits = '01234567890abcdefABCDEF'
```

Implementation dependent values are fields of the global variable *$SYSTEM*, in particular $SYSTEM.epsilon, see System variables and parameters.

It is good practice to start literal names with a capital letter if they are to be exported, while those used locally start with a small letter.

Named literals can be predefined by compiler options: -#MAIN defines #MAIN as ?+, and -#x=5 define x as integer 5.

These definitions are valid from the first line on and may be overwritten by the assignment form.

If the the named literal is not set, it is void, and thus false (?-) in boolean expressions, 0 for integer additions (and subtractions) and void elsewhere.

A few ones (all captical letters) are set by the compiler upon start, reflecting compile time information:

```
#DATE = '2023-01-31'
#TIME = '23:33'
#FILENAME = '...'      \ filename (not path) of the primary input
```

Redefining them is strongly discouraged.

Using the special HTML entity &time; (and &date;, see Debug HTML entities) are the actual value as used in the compile process, but will normally be the same, as the first compilation step (TAV to C) even with large programmes takes only a fraction of a second.

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 maps 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, long function bodies are discouraged, and a rather small number local variables is required so that complex names are not necessary.

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 normally 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 maps); 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). No blank space may follow the dollar sign (it is not an operator, but a name space indicator).

Global variables are shared by all functions in the same compilation unit, 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 variable 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.

A callback on exit may be registered to automate this:

```
main (arg):+
   args =: parse args options with valopts ()
   $opts =: args.options
   system on exit `clear opts` with ()
   ...
\ or with anonymous functions:
   system on exit `() $opts =: ()` with ()
```

Note that parameters are always passed by reference; thus there is no means to pass the address of a global variable to a function, as there is no *address-of* operator and no suitable item kind. (As opposed to functions, for which function references exist).

If there is global data to be shared between modules, it should preferrably be kept and provided in bunches that are made available via mutual function calls.

In case it is unavoidable to share items globally, there is a reserved global variable $COMMON common to all modules, which is initialised with a map 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 neglectable. Of course, fields with module names can be used too. 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 fields, see below.

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 system exit (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.

If threads are provided, global variables will be thread-local, i.e. not requiring synchronisation for access.

Functions use a different syntax for globality.

Local static variables

Because normal globals are restricted to a compile unit, functions that are singletons can use a global variable to maintain their state. Name clashes are manageable, because they are restricted to the compilation unit. The singleton will normally register a function on exit to clear the global. Thus, it is important that more than one function have access to the static variable.

Thus, static variables local to a function are more difficult to clear at end and thus are not (yet) considered.

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:

```
=:  Normal assignment
=~  Set if void
=_  Append expression as string
=+  Add expression to target
=-  Subtract expression from target
=*  Multiply target by expression
=/  Float only: divide target by expression
=!  proxy assignment
```

The remaining ones are partially reserved for future use:

```
=%  reserved for modulo assignment
=^  reserved for exponentiation assignment or proxy dereference
=?  reserved for fault bubble assignment
=&  reserved for boolean and
=|  reserved for boolean or
=<  reserved for getter callback assignment
=>  reserved for setter callback assignment
=,  reserved for tuple extension
=.  alias for '=:'
==  should not be used (ambiguity with equality comparison)
=#  reserved for scan function assignment
=@  reserved (classes and the like)
```

All are digraphs that start with an equals sign, and all supported such digraphs are assignments, that's why the ALGOL assignment digraph `:=` is not used.

The assignment symbol `=:` was clearly inspired by ALGOL. However, the colon is now in TAV-PL an indicator for fuction related operations, and since the implemenation of class functions using the double colon `::` symbol, a clash of colons makes the lines harder to read than necesary. It is thus deprecated in favor of `=.` to distinguish it still from the simple equality operator `=`.

Reluctantly, the equals sign is recognised as a plain assignment symbol alternatively to `=:` and `=.` even if not in C mode, because:

- There is no conflict possible, as assignments yielding values in expressions are not possible.
- When using class functions, the large number of colons makes the programme hard to read
- It seems to be inevitable.

The left hand side is called a reference term, which allows to store item references, and the right hand side is a value expression, 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. Note that the digraph is not `+=`, as all assignments start with an equals sign. In other languages, `x=+1` means `x=:1`, i.e. sets x to +1. For clarity, always use a blank to delimit a bigraph.

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 string catenation converts numbers to strings in default format.

A frequent pattern is:

```
? x = () :
   x =: y
```

The set if void operator is shorter and avoids typing the target twice:

```
x =~ y
```

Currently, the expression y is evaluated, even if x is void; that may change in future, so that if x is not void, the expression y is not evaluated at all:

```
a =: 1
a =~ 2 + print "Must not print"
```

While a field may be the target, a list is not allowed due to lack of clarity.

Note that the inverse digraph ~= is also used and denotes *not equal* comparison.

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 currently empty blocks are not supported. The typical do-nothing-except-advance loop like

```
for (i=start; i!=0; i=i.next)
   ;
```

is anyhow not possible this way and is written as:

```
x =: start
?* x ~= ()
   x =: x.next
```

An empty block is seldomly used, because the loop statement does not directly support statements to advance.

If a scan is to be repeated until it ends, the block repeat ?^ is the logical choice instead of an empty block.

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. However, in the latter case, the layout is not universally equal.

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 is error prone.

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

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

While ?~ is the logical choice, the vertical bar| is often used (not only for historical reasons):

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

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. While this seems to be a waste of vertical space, it is necessary to define the indentation of the following block, and finally requires less lines than the traditional version when swivelled braces occur on lines alone for better readability.

A contraction of a guard in the *else* path is started with the bigraphs ~? or |? (*elif* in some languages):

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

```
   m =: "equal"
```

In C mode, the word elif must be used, not two words else if like in C.

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 might read:

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

Due to code generation deficencies, breaks and repeats currenly may not be used, are not rejected by the compiler and result in assembly errors.

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 map 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 and scan functions

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 key must be repeatedly written.

A scan guard allows to write the same loop as:

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

or with a function similar to Python's `range()`:

```
?# i =: span 1, 10
   print i*i
```

An assignment with a function call at the right hand side is used instead of a boolean expression, and scan functions with specific side effects (similar to generators, a variant of a coroutines) control the scan as follows:

A special global variable, the scan context (denoted as `$`) is initally set to void. For each iteration, the assignment is processed and the function on the right hand side is called. The result is as usual assigned to the variable given on the left hand side. Then the scan context `$` is inspected: if it is not void, i.e. set by the scan function, the body is processed and the loop repeated by perfoming the assignment again.

The scan function saves the context required for the next call in the scan context variable `$`:

```
from (first) upto (last):
   ? $ = ()          \ first call?
      $ =: first     \ set context
   |
      $ =+ 1         \ else advance
   ? $ <= last       \ check for end of loop
      $ =: ()        \ end, clear context
      :> last + 1    \ and provide the next value finally
   :> $              \ return new (current) value for the next round
```

The returned value is assigned to the variable even if void scan context indicates end of loop and the loop body is skipped. This is needed for fault items that terminate a loop prematurely. Also, the next value is helpful in other situations.

The scan context is saved to a hidden local variable each time a scan loop is begun, thus scans can be nested to any depth allowed by the stack, including scan functions inside scan functions.

The above scan function works for all kinds of numbers, as long as they may be used together in expressions, so you may use:

```
?# f =: from 1.0 upto 2.0 step 0.1      # floating point
   ...
?# q =: from 1/10 upto 1 step 1/10      # rationals
   ...
```

Scan functions are often used to enumerate the keys (indices) for rows and maps, e.g. for rows as folllows:

```
row (r) give keys:
   $ =~ r.first - 1       \ start if void
   $ =+ 1                 \ increment the index
   \ find next non-void cell
   ?* $ <= r.last
      ? r[$] ~= ()        \ non-void cell found
```

```
        :> $          \ return its key
    $ =+ 1            \ next key
  $ =: ()             \ scan termination
  :> ()               \ yields nothing
```

The scan loop assignment has on its right hand side an expression that is evaluated exactly like any other assignment, just before the first body statement:

```
?# i =: from pos+1 upto x.Count
  ? c[i] = ';'
    ....
?# y =: from x upto (x+m-1) step inc
  ...
```

As the scan assignment is evaluted each time, the parameters passed may change with each call. It is a question of scan function design whether the parameter values are saved in the context, or used as provided by each call. Normally, the parameters will be literals or variables not changed inside the loop. Note that as in any assignment, the left hand side variable(s) have their old values before they are finally changed.

In the following example, the parameter of the scan function is provided by a costly function call resulting in quadratic runtime, although the result of this function call is be the same each time 8if the set of keys is not changed):

```
\ inefficient (quadratic runtime with the number of keys):
?#  i =: array (array (map m keys) sort ascending) give keys
  ...
\ better in case that the map (m) is not changed in the loop:
mk =: array (map m keys) sort ascending
?#  i =: array mk give keys
  ...
```

The same inefficiency may be caused by calling an expensive function within the loop body. Thus providing a special assignment that caches the parameters of the function would only solve the case in the loop variable assignment, not in the loop body.

The function called in the scan assignment needs not to work with the scan context$ directly; it may use such a function:

```
map (@) give values:+
  k =: map @ give keys
  :> @{k}
```

This general freedom allows some seldomly used scan function wrapper (already in the standard library):

```
give until void (val):+
  $ =: val
  :> val
\ usage:
  ?# line =: give until void (command stream fd get)
    ...
\ instead of
  ?*
    line =: command stream fd get
    ? line = ()
      ?>
    ...
\ unless the scan is already available:
  ?# line =: command stream fd give lines
    ...
```

The (missing) scan function is quite simple and could be defined locally, if not yet available:

```
command stream (fd) give lines:
  $ =: command stream fd get
  :> $
```

A quite sophisticated example just for demonstration:

```
a, x =: 2.0
?# x =- give while positive (x - a/x) / 2.0
  ()
! x %= math sqrt a            \ use near equality for floats
```

instead of the plain version:

```
a, x =: 2.0
?*
   dx =: (x - a/x) / 2.0
   ? dx <= 0
      ?>
   x =- dx
! x %= math sqrt a
```

If a scan function is available, but not as a variant that returns a row or tuple, this can be done by a general utility, that collects the results in a row (or tuple):

```
new row using (func) with (arg):+
   rv =: new row
   ?# v =: func:(arg)
      rv[] =: v
   :> rv
\ fill a row with integer from 1 to 100:
   r =: new row using `give numbers ()` with 1,100
```

As a tuple is passed as a tuple for the (only) parameter, the corresponding function for two parameters uses two parametes without intervening word:

```
new row using (func) with (arg1) (arg2):+
   rv =: new row
   ?# v =: func:(arg1)(arg2)
      rv[] =: v
   :> rv
```

Leaving the loop body by a return or break is normal and supported; however, the scan function will not be notified; its whole context is in the scan context `$`, which will be released by the loop exit.

Because scan functions are lightweight coroutines, several sequences may be generated, e.g. Fibonacci numbers. A tuple is used to hold two values in the context variable:

```
give Fibonacci numbers upto (n):
   $ =~ 0, 1              \ initial tuple if first call
   r =: $.1 + $.2         \ next number
   $ =: $.2, r            \ save previous and current value
   ? r > n
      $ =: ()             \ end of scan
   :> r                   \ return next number in any case
```

An example for a scan in the body of a scan function (a row with a variable field is used as scan context):

```
give primes upto (n) :
   ? $ = ()                \ first call
      $ =: [n]             \ table preset with void
      $.idx =: 3           \ start here at next call
      :> 2                 \ suppply first prime
   p =: $.idx              \ continue here
   ?* $[p] ~= ()           \ skip non-primes, always finds void
      p =+ 2
   ? p > n                 \ done?
      $ =: ()              \ yes, clear scan context
      :> ()
   $.idx =: p + 2          \ prime found, save continue position
   ?# j =: from p*p upto n step 2*p
      $[j] =: p            \ mark all multiples as non-prime
   :> p
```

Using the loop context `$` in the assignment starting the loop is the new context and should issue a warning, unless the code generation is changed. If needed, it must be set aside before, which does not cost any performance:

```
m =: $
?# i =: give keys m
   ....
\ instead of
?# i =: give keys $     \ do not use
```

It is possible to use scan functions inside scan functions outside of a scan loop. The probably most simple example is the value scan of a map (or row) using the key scan:

```
map (@) give values:
   k =: map @ give keys
   :> @{k}
```

```
map (@) give pairs
   k =: map @ give keys
   :> k, @{k}
```

If the function transforms the scan output, the scan context need not to be considered. An exception is the termination in case of the returned value is not possible for tranformation; then, the scan context must be checked for void (only for explanation):

```
map (@) give values:
   k =: map @ give keys
   ? $ = ()
      :> ()
   :> @{k}
```

Another example might be the task to return the values of a row pairwise:

```
row (r) pairwise:
   a =: row (r) give values
   b =: row (r) give values
   :> a, b
r =: [1, 4, 9, 16, 25, 36]
?# p =: row r pairwise
   print p
print "done"
```

The first three pairs are ok, but then the scan continues with `()`, `1` and two more pairs, delivering six pairs instead of three. The cause is that the scan function must be called a 7th time to close the scan in the first call, and with a void context start over again.

So there must be termination check after the first call, and an assertion for even number of elements is useful:

```
row (r) pairwise:
   ! is r.Count even
   a =: row (r) give values
   ? $ = ()
      :>
   b =: row (r) give values
   :> a, b
```

If scan functions are called mistakenly outside an intended loop context, the loop variable is the singular immutable item called `invalid`, so its use in an expression will be detected.

The use of a global variable and the fact that scan functions are not distinguished at all may be considered dangerous. As scan functions should not perform global modifications, such errors will seldomly lead to loss of data. It is anyhow good practice to have plausibility check, e.g. use bunches with tags and check these, or use a check field with the own function reference.

The loop variable may be supplied with a pair, but then parenthesis are required:

```
   ?# i =: from 1,1 upto x,x
      ...
   ?# i =: (from 1 upto x), x
      ...
The single '$' does not conflict with global variables,
because it global variable name have a letter following immedately,
and in particular not a blank.
It refers to the singular item 'invalid', which will inhibit any
use except passing it, e.g. in an expression.
```

The scan function can support calls outside a loop context, and e.g. deliver a tuple — which might be very memory consuming — by using the predicate `is $ invalid` at its entry.

Row, Map and Tuple scans

To print all element values of a Tuple, row and map literals, a range scan may be used, as all keys are integer numbers:

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

If cells have void value, the keys are delivered too; the standard row scan already suppresses these keys (even if the row was modified in the scan body):

```
?# k =: row r give keys
```

```
   ! r[k] ~= ()
```

In several cases just the values are needed; the corresponding scan delivers no valid values too:

```
?# v =: row r give values
   ! v ~= ()
   print v
```

Maps may have any item as key and have no inherent ordering of keys, hence a library function is required to obtain the set of keys actually used (unsorted, normally in last-recently-used order):

```
?# k =: map m give keys
   ! m{k} ~= ()
   print m{k}
```

and corrspondingly

```
?# v =: map m give values
   ! v ~= ()
   print v
```

As indicated, the values for the keys are not void; but if the map is modified in the scan body, this holds no longer, because — for perfomance reasons — the set of all keys is saved initially and then only the keys supplied, equivalent to:

```
rk =: map m keys    \ a row of keys
?# k =: row rk give values
   print m{k}
```

To protect the map from being modified, it can be locked:

```
map m lock
?# k =: map m give keys
   ! m{k} ~= ()
   print m{k}
map m unlock
```

Experience shows that locking as standard measure — although without performace degradation — is to restrictive.

The map scan is provided by the following function:

```
map (m) give keys:
   ? $ = ()
      $ =: map m keys         \ as a row
      $.i =: 0
   $.i =+ 1
   ? $.i > $.last
      ! $.Count = 0           \ all cells already cleared
      $ =: ()                 \ should be totally empty
      :> ()
   rv =: $[$.i]
   $[$.i] =: ()               \ no longer used, chance to free
   :> rv
```

Because the implemetation uses a LRU queue for the keys, the actual order may change even by reading, thus there is no chance to internally use the n-th key in the queue. Note the second-to-last statement, that sets the row entry void, so that the item can be freed early.

The standard row scan is fairly simple :

```
row (r) give keys:
   ? $ = ()                \ 1st call
      $ =: r.First         \ scan context is next key
   \ find next non-void entry
   ?* $ <= r.Last
      ? r[$] ~= ()         \ non-void cell found
         :> $              \ return its key
      $ =+ 1               \ next key
   $ =: ()                 \ scan termination
   :> ()                   \ yields nothing
```

Thus keys for actually non-void cells are supplied in ascending order; if a not yet scanned element is set, it will be found; if it is voided, it will be skipped; also, if cells are added at the end, because the actual .Last attribute is used. Changes on cells with lower indices as the current one do not influence the scan.

If the keys or values are comparable, they may be sorted in advance:

```
rk =: map m keys
rk =: row rk sort ascending     \ choose among the various flavours
?# k =: row rs give values
```

Keys of maps are unique.

Returning bunches in scans

A scan function may well return a row, map or tuple, but this may give unexpected results if the context variable is set to a map which is also returned as the result, as in the following example of supplying fibonacci numbers in pairs:

```
give fibo pairs upto (n):
   ? $ = ()
      $ =: {}
      $.a =: 1
      $.b =: 1
      :> $
   t =: $.a + $.b
   ? t > 10
      $ = ()
      :> ()
   $.a =: $.b
   $.b =: t
   :> $
test printing:
   ?# m =: give fibo pairs upto 10
      print m.a __ m.b
test saving:
   r =: []
   ?# m =: give fibo pairs upto 10
      r[] =: m
   ?# i =: r.scan
      m =: r[i]
      print i _ ': ' _ m.a __ m.b
main (parms):+
   test printing
   test saving
```

The result will be:

```
1 1
1 2
2 3
3 5
5 8
1: 5 8
2: 5 8
3: 5 8
4: 5 8
5: 5 8
```

So in the second scan, only the last pair is shown. The reason is that the scan returns always a reference to the same map, so the row cells all refer to the same map and print the same (last) values.

Scans returning immutable items like tuples, numbers and strings do not have this problem, as the items are created afresh each round. The above could be repaired by returing a copy of the map:

```
:> map $ copy
```

This phenomenon is one of the reasons why tuples are conceptually immutable; however, the following code works, because not the tuple elements are overwritten, but the tuple replaced by a new one:

```
give fibo pairs upto (n):
   ? $ = ()
      $ =: 1,1
      :> $
   t =: $[1] + $[2]
   ? t > 10
      $ = ()
      :> ()
   \ $[1] =: $[2]; $[2] =: t is not allowed
   \ instead, create a new tuple
```

```
    $ =: $[2], t
    :> $
```

Unless the returned tuples are saved in a row, they are freed after print (more accurate: when the value of the loop variable is overwritten from the scan function and its use count goes to zero).

2.8. Operators and Expressions

Expressions

Expressions deliver values, i.e. item references, derived from literals, variables and function invocations, either directly or transformed by operators or function invocations and combinations thereof.

Expressions are sequences of terms, operator symbols, function invocations and lists, where terms are:

- literals (numbers, strings, …)
- variable names
- function calls
- expressions enclosed in parenthesis

Operators are either monadic, i.e. written left to a term that is the only operand, or dyadic, i.e. between terms, that are the left and right operands. The kind of the result can be determined from the kinds of the operands independent of the actual values, once the latter are known.

Any operator (as well as a function call) works as if their return value replaces the operator and its operands. The evaluation of an expression is equivalent (and most often done this way) to a series of assignments of an operator and its operands to temporary variables, and then using the temporary variables replacing the operator and its operands.

Very often, the order in which the operators and functions calls are to be applied must be determined, which is done by assigning priorities to them. It is well known as a rule that multplications have priority over additions, if in doubt. In the following examples, the order is indicated by using (redundant) parenthesis:

```
a + 5 * b       ==  a + (5 * b)
a * 7 * -b      ==  (a * 7) * (-b)
- -a            ==  - (-a)
```

The priorities are descending (highest first):

- Monadic operators except boolean negation
- Arithmetic operators in common priority order
- String operators
- Comparison operators
- Boolean operators with boolean negation first

The first three never provide logical results, the comparisons use numbers and strings and give boolean values, and the boolean operators use boolean values to produce boolean values. This is mirrored by the priorities.

In traditional denotation of function invocations, parameters are expressions or lists of expressions, enclosed in parenthesis. In TAV-PL, function templates (see [function call]) use words to delimit parmeters, so there are no parenthesis required if the parameters are terms:

```
?# i =: from 1 upto x.Last
   ....
\ instead of
?# i =: from (1) upto (x.last)      \ valid, may be used for clarity
```

Allowing expressions instead of terms as parameters is only natural and forces the use of parenthesis infrequently:

```
?# i =: from k + 1 upto x.Last
   ....
\ instead of
?# i =: from (k + 1) upto x.Last
   ....
```

As the word upto already delimits any expression (no valid expression ends in a term followed by a word), the scan for parameters extends to expressions, reducing the number of parenthesis needed.

The alternatives are:

- Use parenthesis as parameter delimiters (always required):
  ?# i =: from (1) upto (x.Last)
- Only allow terms without parenthesis:
  ?# i =: from (k+1) upto x.Last

The *greedy* expression scan is quite simple a rule, and thus is a good basis for audits, but may occasionally surprise.

Consider some assignments (func is a function with one argument):

```
x =: a + b
y =: func a + b        \ = func (a) + b = func (a+b)
z =: (func a) + b      \ parenthesis required
x =: func a + 1 + b    \ = func (a + 1 + b)
y =: (func a) + 1 + b
z =: (func a + 1) + b
```

Because parenthesis are not delimiters for function parameters (unless there were such a compiler option or pragma), limiting function parameters uses the unusual notation that the whole function invocation is enclosed in parenthesis.

Extending the parameter scan even to lists, and because list elements cannot be lists, makes the print function quite comfortable, because it accepts tuples:

```
print x, y             \ = print (x, y)
print x + 1, y         \ = print (x+1), y
print x + (1, y)       \ addition must accept tuples
```

As function parameters may be lists, the use of function results in the list of print parameters requires parenthesis:

```
print (group x), group y
print group x, group y     \ = print group (x, group y)
```

The last line works fine, because the function group is idempotent and accepts lists, however the following example fails unless the square root accepts tuples:

```
print math sqrt 2.0, math sqrt 2.0     \ fails
print (math sqrt 2.0), math sqrt 2.0   \ prints "1.41421 1.41421"
```

As lists include logical expressions, note the following two statements:

```
print (math sqrt 2.0) < 5.0    \ prints ?+
print math sqrt 2.0 < 5.0      \ fails: not sqrt from bool
```

Operators

Because the kind of operands is not known at compile time (unless the constaints system is functional), operators determine at each invocation if the operands can be processed. If not, either void is returned, or an error is signalled.

Note that bigraph symbols are composed of two special characters without intervening spaces.

List of symbols used:

```
?      Guard (if)
|?     Guard alternative (elif)
|      Guard default (else)
?~       "     "
?*     Loop
?#     Scan (scans, iterations)
?^     loop repeat
?>     loop break
??     Error Guard
?|     Error catch
?(     conditional expression begin
?!     conditional expression true part begin
?:     conditional expression false part begin
?)     conditional expression end
=:     Assignment
$      Global variable name prefix
#      Named literal (global constant) prefix
!      assertion and constraint
:      functions
:>     return function value
```

```
:(     reference call
::     class function definition and call
```

Instead of special characters, (reserved) words may be used if enabled by the pragma \C:

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

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:

```
->  pair (tuple with two elements) creation
+   addition
-   subtraction
*   multiplication
/   division in general
//  integer division with truncation (for accurate numbers, see [[Integral Numbers]] for variants)
%%  integer modulus (see [[Integral Numbers]] for variants)
^   exponentiation
-   (unary) sign change
##  (unary) convert to floating point (also as attribute and function)
*   (unary) proxy dereference
&   (unary) proxy obtain
```

To obtain the absolute value or the sign of numeric items, see the respective descriptions.

Basically, the operators return the same kind as the operands and require both operands of the same kind, with the exceptio that integer and rational may be mixed, because conceptually integer numbers are a subset of rational numbers. The intention is to avoid accidential loss of precision for reliable programming.

Unless disabled by the \a pragma, newer versions provide the automatic conversion of integer (and rational) operands if the other one is a floating point number, then returning the latter kind. While this implies generally a loss of precision, the result is always a floating point number if one operand is such a kind, which is known to be an approximate number.

In the accurate mode (via the\a pargma), an accurate number (integer or rational) has to be converted to a floating point number (if the other operand is the latter) using the ## operator, the .float attribute or the integer (@) as float function (or rational (@) as float). To apply the .float attribute to expressions, in particular integer literals, parenthesis are required. The ## operator, although ugly, has the advantage of the high precedence of a unary operator, but might become deprecated. The distribution function as float() might require parenthesis; better use the class function ::as float:

```
i =: 3
print ##i * 2.0
print i.Float * 2.0
print integer i as float * 2.0
print (as float i) * 2.0
print as float i * 2.0              \ fails: as float (i * 2.0)
print i::as float * 2.0
print 2.0 * integer i as float
print 2.0 * i::as float
print ##i, 2.0
```

As the integer numbers are not limited, they may exceed the range and provide a NAN or return a fault item, which normally terminates the programme.

See the Attributes section for details on this and the reverse conversion.

The exponentiation operator (^) is quite flexibly implemented:

- If the base is integer, the exponent must be a non-negative integer
- If the base is 2 and the exponent small, a shift is used
- If the exponent is 2 or 3 and the base small enough, multiplications are used
- otherwise, the multi-precision library is used for integer exponentiation.
- If base is float and the exponent is one of the integers 0 to 4, multiplications are used;
- otherwise, the standard C library is used, and the base may not be negative, as $-2^{0.5}$ would result in a complex number (note that the library may multiply the logarithm of the base by the exponent).

So the exponentiation of a float by an integer is possible even in accurate mode, in contrast to the general rule that mixing kinds in operators is not possible.

For tuples, the arithmetic operators are available too, working element by element. Thus, the kinds of the elements must match for the opertors.

The combination of a tuple with a plain number is possbile to, as if the plain number would be expanded to a tuple of the required length. A tuple is returned in any case, in particular if a tuple is subtracted from a plain number:

```
t1 =: 1, 2, 3
t2 =: 1 - t1
! t2 = 0, -1, -2
```

The maximum and minimum fuctions might some day be extended to allow a tuple as a member of a tuple, in which case any other tuple must have the same size, and the operation is done elementwise.

No such operation is possbile for rows.

If a void operand in an addition would be a fatal runtime error, using an element of a map as counter would be coded as:

```
? map{idx} = ()
   map{idx} = 1
map{idx} =+ 1
```

To avoid the two leading statements and make the programme easier to read, a void reference is treated as number zero in additions, thus the leading two lines are not be necessary. 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.

Performance is not increased significantly (except saving two instruction times), as in both cases the map is searched two times for the not yes existing key: one to find it is void, the second one to set it. The third access above is quick, as the last found index is cached.

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 strings 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)
=%  reserved (remainder)
=^  reserved (exponentiation)
```

The symbols are intentionally inverted over the form known from the C language and its siblings, where `x=+5` means `x = +5`, because blank space is not relevant. 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.

The shortcut =% for *divide by rhs and supply the remainder*is not supported, due to the three different version of remainder calculation.

The =/ operator is limited to floating point numbers.

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

Note that no arithmetic or boolean 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
<=  less or equal, not greater
~<  not less, greater or equal
>=  greater or equal, not less
.=  equal kind
~.  unequal kind
@=  equal class
~@  unequal class
?=  guarded equality
~?  guarded unequality (for assertions)
%=  near equality for floats
~%  near unequality for floats (for assertiona)
^=  not yet used (may be proxy reference check)
!=  reserved as alias for '~='
==  reserved as alias for '='
<>  reserved for ~=
><  reserverd for ~%
#=  nor yet used
_=  not yet used
+=  reserved (used in C differently)
-=  reserved (used in C differently)
*=  reserved (used in C differently)
/=  reserved (used in C differently)
```

In a bigraph using the equals character (=), it is always the trailing character (assignments start with an equals character); the corresponding negation starts with the tilde (~) and is followed by the character before the equals character. Thus, there are no overlaps of arithmetic operators, logical operators and assignment statements. (An confusion with the assignment digraphs is anyhow not possible, as the target of an assignment is not an expression.)

In order to detect errors early, the comparisons are restrictive and can fail in a couple of situations, which normally is a fatal runtime error, leading to an immediate error stop, depending on the ??? fault mode.

The rules for equality comparison are:

- equal references yield true without further inspection (including void)
- void may be compared to any other reference yielding false and is never an error,**not** treated as zero
- Integers compared to integers use the numerical value.
- rationals compared to rationals use the numerical value
- Integers and rationals yield false (without error) as rationals are automatically converted to integer if the denominator is 1.
- Floats may only be compared for equality to the special values #+ for +∞, #- for -∞, #~ for a quiet NaN and #! for a signalling NaN. Otherwise, comparing floats for equality is an error.
- Strings are compared byte-by-byte, which is the same as character-by-character for UTF8 strings.
- Tuples with different number of elements are never equal; with the same number, the items are compared by guarded equality, with the exception that floats are never equal, even if nearly equal.
- Other items, in particular rows and maps, must be of the same kind and are never equal (unless the references are the same).
- If one operand is a fault item, a new fault item is returned, with the other operand saved as reference in the data field (to be implemented).
- Items of equal kind are not equal unless the references are the same.

- Otherwise, it is an error comparing different kinds.

The last case can be avoided by using the guarded equality (?=), giving false unless both items are of the same kind and are equal:

```
a ?= b  ≡ ?( a.Kind = b.Kind ?! a = b ?: ?- ?)
a ?= b  ≡ a.Kind = b.Kind && a = b    \ see below
```

As this operator is intended to never be faulty, and equality comparison of floats is an error, comparison of two floats is always false:

```
a ?= b  ≡ a.Kind = b.Kind && ~is a float && ~is b float && a = b
```

To check if an increment would change the value, it is be necessary to check it is not void first:

```
? a = () || a ?= 0 || a %= 0.0
   \ action if neglectible increment.
```

Void is **not zero** in comparisons so that comparing the integer 0 with a map or row cell allows to distinguish void and the number.

Key comparison for maps is similar, but floats are erroneous.

Because equality comparison for floating point numbers is erroneous, there is a function to test near equality:

```
is (x) near (y):
   :> is x near y by ()
is (x) near (y) by (d):
   ! is x float && is y float && is d float
   d =~ 5 * $SYSTEM.epsilon        \ default value
   x =: math abs x
   y =: math abs y
   max =: maximum of x, y
   diff =: math abs x - y
   :> diff <= max * d
```

The operators %= and ~% provide near equality aud unequality, the latter often used in assertions.

Order comparisons <, >, etc. use only the *greater* comparison, with negation if necessary:

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

They follow the rules for equality comparison, except:

- Comparison with void is always false
- Floats are compared to floats without error
- Integers or Rationals with Integers or Rationals are compared numerically without error
- Tuples with equal numbers of elements are order compared element by element until one is greater than the other, or not equal. If the number of elements is not equal, the one with more elements is greater.
- Other items of the same kind yield false, i.e. are never greater.
- Otherwise, comparing different kinds is an error.

Void is not a number and thus need not follow the mathematical rules of number comparisons; it is zero in additions only for convenience.

Tuple order comparison is primarily intended for sorting; using pairs as complex numbers is not compatible, as complex numbers do not allow ordering. For sorting, consider the functions tuple (p) greater (q) cell (n) etc. that allow to compare one element only or a different order of elements to compare.

Although the alias digraphs suggests it, no test for equality is done for>= and <=. Using x <= y & x >= y is equivalent to ~(y > x | x > y) , so there is still no equality compaison generated; and comparison of floating point numbers is always possible.

The *equal kind* comparison operators .= and ~. could compare the kinds of two items, or one with a string. They are deprecated because seldomly needed; class comparisons are more often requied, and if needed, it is clearer to express the situation:

```
? x.Kind = y.Kind       \ compares only .Kind, not .Class
? x.Kind = 'integer'    \ use next line instead
? is x integer          \ supplied for every kind
```

```
? is x hashmap        \ class comparison provided by hashmap library
```

The *equal class* comparisons @= and ~@ originally were tag comparisons and are deprecated for the same reasons as for the kind comparisons; a class comparison of two variables is simply

```
? x.Class = x.Class
```

The reserved global variable $SYSTEM provides the field $SYSTEM.epsilon, which is the smallest difference between 1.0 and the next larger number. Note that this is not directly applicable as (additive) *epsilon*, [4] because it is the smallest such value for numbers around 1.0. It can, however, be used in schemes like the above elaborate comparison.

There is no operator to check if two references are equal because there is no reference item (i.e. reference to a reference); comparing e.g. two integer references this way has no sensible applications.

It is possible that a fault item is the result of an error condition; but then, each operator must not generate another fault item if an operand is such, but pass them upstream.

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).

Maps 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 map elements as counters would require tedious programming, e.g. for counting occurrences of strings:

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

instead of

```
map{str} =+ 1
```

When a void key is used to read a row, map or tuple, the result is void. On write, however, it's a fatal runtime error.

Voids for multiplications and divisions are a fatal runtime error; they are neither treated as zero nor as one, in order to keep the rules simple.

Currently, operators (except one, ^=) never return fault items due to historic reasons; first of all, fault items were introduced later, and the sensible use of fault items for operators was unclear; in particular, if an operator has two operands that are fault items.

As TAV-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.

Logical operators

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

```
&   for logical and of booleans
|   for logical or of booleans
~   for logical negation of booleans
=   for logical equivalence of booleans
~=  for logical antivalence (exclusive or) of booleans
&&  shortcut evaluator 'and then'
||  shortcut evaluator 'or else'
```

The implication must be expressed using:

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

The terms in logical operators can be comparisons, 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, including the logical and (`&`) and logical or (`|`), but excluding the shortcut evaluators.

The shortcut evaluators `&&` and `||` are essentially conditional expressions:

```
lhs && rhs ≡ $( lhs ?! rhs ?: ?-  $)
lhs || rhs ≡ $( lhs ?! ?+  ?: rhs $)
```

They are thus not commutative and evaluate the right hand expression only if the left hand one is false (and) or true (or). [5]

If tuples are used as operands for comparisons, they must be in parenthesis.

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

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

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

```
#Letters = 'abcdefghikjklmnopqrstavwxyz'
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 neither the number zero, nor 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
```

The unary logical 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
```

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

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

String expressions

The primary String operators and assignments are:

```
  =   equal
  ~=  not equal
  _   catenation
  __  catenation with a blank between, i.e. '_ ' ' _'
  _>  reserved for right alignment (left padding)
  _<  reserved for left alignment (right padding
  =_  string append assignment
.
```

The left and right padding operators are be not yet implemented; use`pad left (cnt) (str)` instead of `cnt _< str` and `pad right (cnt) (str)` instead of `cnt _> str`.

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
msg =: "Error copying from &source. to &target."
msg =: string msg replace ('&source.', srcfn), ('&target.', tgtfn)
```

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

```
msg =: "Error copying from «source» to «target»"
msg =: string msg replace multiple ('«source»', srcfn), ('«target»', tgtfn)
msg =: msg::replace many (...)
```

The multiple replacement function has the advantage of strict left-to-right processing, i.e. a replacement is final.

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
```

If the item to be catenated is a tuple, its values are delimited by commas and enclosed in parenthesis and, if the elements of a tuple are pairs, the are delimited by `->`. Rows and maps are not catenated, as they may have fields that would not be shown.

Note that the catenation operator _ just returns the other operand if one operand is void or the empty string, so it is perfecty proper to rely on the automatic conversion in

```
v =: 10
str =: '' _ v
```

For debugging, often the following patterns are used:

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

It would be nice to write instead

```
debug print ?p
debug print ?ary{ai}
```

This could be supported by a Macroprocessor, because the current compiler structure parses expressions without setting aside the source text needed here.

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. If the lengths differ, the shorter one is smaller. 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.

As with other items, if the references are equal, no further comparison is required; thus, a string function may return the original string if nothing is changed, and string equality is efficient to determine this case.

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 TAV-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.

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

```
string (str1) compare to (str2) under locale (locale):
\ returns -1 for less, 0 for equal and +1 for greater
string (str1) compare 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 TAV-PL 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. Furthermore, the use of integer numbers as rows of bits is neither intended nor efficient in TAV-PL. Some functions will be implemented in C; the TAV code is shown here for demonstration.

To check whether a number is odd or even, the remainder modulo 2 is taken (both are already defined in the standard library):

```
is (x) odd:
   :> x %% 2 = 1
is (x) even:
   :> ~ is (x) odd
```

Due to the use of the absolute remainder`%%`, the result the same as if the last bit in two's complement is checked. Note this function returns a boolean (as the library function does), if an integer is required, use `x %% 2` directly.

Using arithmetic shifts (that retain the sign) for multiplications or divisions by a power of 2 is a bad habit from ancient times; is is not at all more efficient here, only less clear. And today practically all compilers optimise multiplications and divisions to shifts, if more efficient. All other shift operations use the binary representation of an integer number as a row of bits and thus heavily depend on the number of bits used.

In case it is required not to use integer numbers, but rows of bits more efficiently than using a row of booleans (or integers), a library must be used:

```
y =: bits x and 7
y =: bits x or 4
y =: bits x xor 8
z =: bits x shift 5        \ = x * 2^5 %% 2^31
z =: bits x shift -5       \ = x * 2^-5
print bits x inverted      \ just flip all bits, one's complement
```

Above functions are supported only on (64-bit) machine integers.

To test if a bit is set, the following library function supports big numbers too and treats negative numbers with two's complement:

```
bits (x) is (b) set:
   ! b >= 0
   :> is x // 2^b odd
```

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 is independent of order (note that there is no integer overflow in TAV-PL).

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 & &&
4  b ->
4  b < > <= >= = ~=
5  b _
6  b + -
7  b * / // %% /% etc
8  b ^
9  u - ~ `
```

Literal denoting symbols are not operators, 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 reference and not an operator.

Map and row element selection as well as field and attribute notation are not an operators; they are variable denotations.

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 key have different kind
 ? r.kind ~= s.kind         \ we are not comparing rows and maps
   :> ?-
 ? r.count ~= s.count       \ same number of elements?
   :> ?-                    \ no
 ?# i =: r.scan             \ give all keys of r
   ? r[i] ~= s[i]           \ fatal error if comparison not defined
      :> ?-                 \ unequal
 :> ?+                      \ ok, all keyed fields are the same
```

2.9. Tuples, lists and list assignments

A list is a sequence of elements, separated by commas. It can be used:

- instead of an expression to create a tuple
- on the left hand side of an assignment

In the fist case, a list creates a tuple from the list elements. The second case is described below as tuple assignments.

A tuple is another kind of item; an immutable row:

```
a =: 1, 'x', math sin 5.0
! a.count = 3
! a[1] = 1
! a[2] = 'x'
! is a[3] float
```

Empty list elements are ignored; to create a cell with a void element, it must be explicitely written:

```
l =: 1, ,3
! l.count = 2
l =: 1, (), 3
! l.count = 3
! l[2] = ()
```

In order to avoid accidential creation of a list by stray commas, leading and trailing commas do not create a list; `x,` as well as `x,` do neither create a list with one element nor are they just the reference.

To create a tuple with no or a single element, use the factory fuction:

```
t =: new tuple size 0 values ()
t =: new tuple size 1 values 2
t =: new tuple size 1 values 1,2
! t.Count = 1 && t.1 = 1,2
```

It had been tempting to tweak the rules such that a trailing comma in a print function call would substitute the trailing NONL, e.g. by producing a void element. But a similar rule in IBM's Job Control Language did cost enormous amounts of money in the 1970's, and has thus been banned because it is not visible enough.

To access elements, either the same notation as for a row, i.e. the key in square parenthesis, which may be a integer variable or literal, or field notation with a literal integer (not variable) instead of a word:

```
t =: 1, 2
! t[1] = t.2 - 1
```

To call a function with a variable number of parameters, use a single argument and a tuple when calling it:

```
maximum of (tuple):
   ? is tuple tuple
      mx =: tuple[1]
      ?# i =: from 2 upto tuple.count
         ? mx < tuple[i]
            mx =: tuple[i]
      :> mx
   :> tuple
print maximum of 4, 6, 5          \ prints 6
```

As tuples are immutable, to select a slice or append an element always has to return a new tuple with the selected elements; their implementation may still be inefficient:

```
tuple (@) append (val):
tuple (@) from (from) upto (upto):
```

It is nearly always more efficient to use a row in the first place, and finally — if necessary at all — obtain the equivalent tuple.

There are some cases with lists, which need clarification, e.g.:

```
:> maximum of a, b, c
\ which is
:> maximum of (a, b, c)
\ and neither
:> (maximum of a), b, c
\ nor
:> (maximum of a, b), c

:> print group a, b, group c
\ which is
:> print (group a), b, group c
\ and not
:> print (group a, b, group c)
\
```

The print function accepts tuples too and separates the parameters by blanks (and converts numbers to strings):

```
print (arg):
   ? is arg tuple
      sep =: ''
      ?# v =: give arg values
         print sep _ v nonl
         sep =: ' '
      print ''
   ? is arg string
      print arg
   |
```

```
    print number arg as string
  print"

print 4, 5, 6
```

If this is not intended, use the join function:

```
x =: 4, 5, 6
print join x by ''      \ == print '456'
```

Tuples can be created anywhere by a comma separated list of item references. Parenthesis are neither required nor necessary; but may be used to clarify interpretation.

Because the comma separates expressions,

However, on a parameter postion that is not the last one, the parenthesis are required:

```
print math sqrt 4, '=', 2
\ same as:
print math sqrt (4, '=', 2)
\ but probably meant was
print (math sqrt 4), '=', 2
```

Because a list is composed of expressions, the comma stops expression scan and avoids to add an integer to a tuple:

```
print 4 + 5, 6              \ prints 9 6
print 4 + (5, 6)            \ not (yet) supported
```

When calling a function, a list may be used as parameter, not just an expression, thus the second line is the equivalent of the first one:

```
print maximum of 4, 6, 5        \ prints 6
print maximum of (4, 6, 5)      \ prints 6
print (maximum of 4), 6, 5      \ prints 4 6 5
```

To allow a lists of variables in a function template as in:

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

might be useful if a fixed number of similar parameters is required. As in most of these cases, a variable number of elements is more natural, this notation is not supported.

Quite often, a tuple is returned from a function:

```
do something:
  ...
  :> code, value
...
cv =: do something
? cv.1 = code_x
  process cv.2
```

As a tuple is immutable, a tuple element cannot be used on the left hand side of an expression; the whole tuple must be created again. For example, to rotate tuple elements or exchange two variables, write:

```
l =: l.2, l.3, l.1
x, y =: y, x
```

A tuple may be used as a key for a map, because tuples can be tested for equality (see below):

```
m =: {}
m{1,3,5} =: 135
```

For an array, it could be used as keying a multi-dimensional array via the matrix library:

```
a =: new matrix 10,10
a[5,5] =: 55
```

Using a tuple as a key for a string for picking the indicated characters is considered sparely used, and might confused with a multidimensional array, so this is not supported.

Dynamic tuple creation

Tuples are conceptually immutable like literals, but in particular in libraries it may be necessary to create a tuple with elements determined at runtime.

For this purpose, there are library functions:

```
row (row) as tuple:
   \ create a tuple with all (even void) elements
new tuple (count) values (vals):
   \ new tuple with all cells preset
tuple (tuple) as row:
   \ create a row with all elements of (tuple)
```

The second funtion is in particular useful if a tuple of unity elements is required.

A simple library function allows to create a tuple from a part of a tuple:

```
tuple (tup) from (from) upto (upto):
   new =: row tup from from upto upto  \ Works for tuples too
   :> row new as tuple
```

It is provided just for clarity, as are the following ones:

```
tuple (tup) drop first:
   :> tuple tup from 2 upto tup.last
tuple (tup) drop last:
   :> tuple tup from 1 upto tup.last - 1
tuple (tup) prepend (item):
   r =: tuple tup as row
   ! r.first = 1
   r[0] =: item
   :> row r as tuple
tuple (tup) append (item):
   r =: tuple tup as row
   r[] =: item
   :> row r as tuple
```

Tuple equality

While rows and maps intentionally cannot be compared for equality due to their dynamic nature and unclear merge of fields, comparision of tuples is possible, as tuples are fixed size without fields; thus, the comparison simply must check for equal size, and with equal size do a guarded comparison, i.e. the same used for map keys (floats are always unequal here).

List assignments

A list of variables can be used on the left hand side of an assignment:

```
a, b, c =: 1, 'x', sin 5
! a = 1, b = 'x', c = sin 5
x, y =: 0
! x = 0, y = 0
```

Yet the left hand side is not a tuple item, as this would require references to variables (or bunch elements, etc), it is just a list of assignment targets.

The result depends on the result of the right hand side expression, that is evaluated once:

- If source is not a tuple, it is assigned to all destinations on the left hand side, i.e. copied several times.
- If the result is a tuple, the elements are assigned to the left hand side destinations in normal key order 1, 2, 3 …. If the tuple is exhausted, void is assigned.

If there are items left over, they are ignored silently:

```
t =: 1, 2, 3
a, b =: t
! a = 1 & b = 2
```

The tuple assignment

```
a, b =: RHS
```

is a shortcut for:

```
t =: RHS    \ right hand side, tuple or other item
? is t tuple
   a =: t[1]
   b =: t[2]
|
   a =: t
   b =: t
```

The source item may be in the target list, as in:

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

If the same variable is used more than once in the target list, it is simply overwritten:

```
a, a =: 1       \ same as a=:1; a=:1
```

However, if the source is a tuple, it is expanded on the right hand side, but not on the left hand side; thus a tuple on the left hand side is replaced by whatever element was used on the right hand side, which might be not as tuple:

```
t =: 1, 2, 3
t, x =: t
! t = 1 & x = 2        \ t is now t[1] and x is t[2]
t =: 4, 5, 6
t, t =: t              \
! t = 5
```

In order to avoid hard to find errors, if the source is a tuple, it is protected as shown above before the individual assignments take place.

In the following example, the random number is obtained only once:

```
a, b =: random next integer
! a = b
```

To exchange two values, a tuple assignment cat be used, because first the source items are collected into a tuple, then assigned to the target variables:

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

is equivalent to

```
t =: b, a
a =: t[1]
b =: t[2]
```

It might be tempting to assign the last variable a tuple with the remaining elements if there are less destinations on the left hand side than tuple elements on the right hand side.

The disadvantage is that in most cases the last variable on the left hand side must be checked whether it is a tuple or not, and thus destroying the terseness of the construction. It would add complexity without proved gains in clarity and reliability.

Tuple assignements are not necessarily more efficient than several assignment lines,

3. Items

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

3.1. Fields

Because attribute fields of items are used in the description of the various kinds of items, they are introduced first.

Fields concern information for items, named at compile time by words (letters followed by letters and numbers). The name of a field follows the item variable, attached by a point (full stop, .), e.g.:

```
var.Kind
```

There are three types of fields:

- variable fields (fields)
- attribute fields (attributes)
- control fields (controls)

Fields of literals etc. are quite senseless; the can be obtained if enclosed in parenthesis to form an expression:

```
! (1).float = 1.0
```

Variable fields are available for mutable items only, in particular for rows and maps. They are variables bound to the actual item dynamically, not statically declared as in strongly typed languages. Organisational information can be attached to bunches (rows and maps) this way:

```
m =: {}             \ create a map
m.longest =: -1     \ keep track of longest entry
r =: []             \ create a row
r.max =: 0          \ to keep track of maximum value set so far
```

Retrieving a not yet set variable field is void, and any such field can be removed by setting it void; after that, it cannot be distinguished from a never set field.

Names for fields generally start with a lower case letter.

The attempt to create a variable field for an immutable item is a fatal runtime error, while trying to obtain such a field is just void without error.

Variable fields are effectively a map with keys that are programme literals; they use a different namespace as the variable keys used for maps and rows.

Attribute fields provide metadata of items; they cannot be directly modified (e.g. by an assignment). Each item (including void) has at least the .Kind attribute, which is a short string:

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

The kind of an item determines which attributes are valid; invalid attributes return void and are not an error.

For immutable items, attributes provide e.g. the number of characters (code points) of a string or the sign of a number; no functions are necessary for such elementary information.

For mutable items, attributes provide information about the actual state of the item (they can change over time); e.g. .Count is the number of (non-void) elements in a row or map.

Attributes use the same notation as fields, but normally start with a capital letter:

```
.Kind .Count .First .Last
```

Attributes are also available for immutable items, like numbers and strings; as they cannot be mistaken as fields.

Attributes of numbers are also lower case (note that numbers are immutable and thus cannot have variable fields):

```
.abs .den .float .frac .trunc .isbig .num .rsd .round .quot .sign .min .max
```

Examples for less commonly used attributes without small letter aliases are:

```
.Bytes .Amount .Usecount .Revisions .Dump
```

There is no protection against using the small letter variants as fields; however, the compiler might warn if these words are encounted as lower case field names.

As attributes are read-only, an attempt to use it as a field will be detected very soon. As attributes are system defined, there is a fixed list, and the compiler can signal an error if target of an assignment.

It has been tempting to provide boolean attributes like .isInteger, .isFault, .isASCII for their brevity, but discarded, as the corresponding functions

```
is () integer
is () fault
is () ASCII
```

are also easy to read, and attribute notation would neither increase readability nor performance significantly. In particular, although originally considered a nice feature, these attributes effetively just check the kind of item, and do not represent a true characteristic of an item.

Control fields are neither variable fields nor (read-only) attributes and start like attributes with a capital letter.

Their content controls the way an item is processed. E.g., setting the control field.Lock for bunches (arrays and maps) inhibits further changes (until reset).

Control fields can thus conveniently used in an assignment:

```
bunch.Lock =+ 1        \ increase lock counter
```

Once field change callbacks are implemented, libraries can control which settings are permitted, so that a field like this is like a control field and will often start with a capitial letter.

A special case is .Tag, that can only be set once, except when the new value is the same as the previous one.

Using variable field names starting with lower case letters is recommended. New language versions might introduce new attributes and control fields, but always starting with capital letters, so there will be no collision with already used field names that start with small letters (except the above commonly used small-letter aliases).

Fault items have a field .checked that tells if the fault item is still unprocessed (not checked). All other information in a fault item is a preset field; for ease of use, all, including the .checked control field, are written in lower case; thus, to mark a fault item checked:

```
fault.checked =+ 1
```

As the field and attribute notation returns any item reference, the attribute .scan could provide the basic key scan function for a row or map. But attributes returning a function reference are deprecated for clarity; use the appropriate scan function. Providing functions to get or set attributes or fields is mostly deprecated now except for those noted below. In fact, if a field is a function reference, its call must be written like any other one:

```
this.sortcall:(x, y)
```

Some attributes are common to all kinds of items:

name	description
.Bytes	size of storage occupied (bytes; 0 for void)
.Kind	kind of item as string (empty string for void)
.Tag	Name of a bunch (row or map)
.Class	Name for any item (related to .Tag')
.Usecount	usecount (0 for void, >1 for all others)
.Dump	Formatted diagnostic information (long) string

The .Kind attribute returns a fixed set of strings for the kind of the item queried:

- boolean
- chunk
- complex
- fault
- float
- integer
- map
- rational
- row
- string
- tuple
- system

There are no truncated forms (bool, int, rat, str or sys), as they are used in templates for indicating the kind of items that are parameters, where the long forms are better readable. Also, this leaves the short words useable for local variables.

To check the kind of a function should be done by the functions.is () ... that are be provided for all kinds in a localised version.

The .Tag control is void for all items initially. It can be set once for mutable items, in particular maps and rows, normally by the creator of the item. Setting .Tag with one of the kind strings is not possible. For system items (.Kind = 'system'), which are library bindings, it is set by the library interface and cannot be changed either.

The .Class attribute is determined as follows:

- If .Tag is not void, it is supplied.
- Otherwise, the value of .Kind is supplied.

It is used to identify different items of the same kind, in partiuclar for [class functions].

The subtle differences between .Kind, .Tag and .Class are normally of no concern. But if only .Class were kept (and allowed to change once), the implementation would still be the same, and the description would implicitly use the concept of .Kind and .Tag, whithout exposing them, which would be overly complicated to describe and thus hard to understand.

The following number converstions can be accessed as functions and attributes.

name	function	description
.abs	attr abs ()	absolute value
.ceil	attr ceil ()	smallest integer greater or equal
.den	attr den ()	denominator (=1 for integers, > 1 for rationals)
.float	attr float ()	closest floating point number
.floor	att floor ()	largest integer less or equal
.frac	attr frac ()	fractional part: (x - x.trunc.float).abs
.int	attr int ()	arbitrary large integer truncation (see below)
.round	attr round ()	nearest integer
.num	attr num ()	numerator (signed)
.sign	attr sign ()	sign (see below)
.trunc	attr trunc ()	truncate, round towards zero (signed)
.wide	attr wide ()	widen, round away from zero (signed)

As usual, the attributes give void if not applicable; so do the functions, that use the same code; but some error conditions do not return void, see below.

For floats, .den, .num and .rsd are void, and .float is just a copy of the reference, as is .trunc and .wide for integer.

The sign .sign is of the same kind as the operand, so that the relation x = x.abs * x.sign is always valid.

If you need an integer sign of x, in particular for non-zero numbers between -1 and +1, use x.sign.wide; it can be used for integers and rationals too.

The truncate and widen attributes are equivalent to:

```
math trunc (x):
   :> x.sign * x.abs.floor
math wide (x):
   :> x.sign * x.abs.ceil
```

To convert a float to an integer, use the attributes .ceil, .floor, .round, .trunc or .wide as required, or their corresponding functions. Floating point numbers above $2^{53} \approx 9.0 \cdot 10^{15}$ (abs) have the decimal fraction zero and would deliver all the same result, showing seemingly random digits after the 15 leading digits. Therefore, using this attributes (and the corresponding functions) for numbers above is an error, not void.

If accuracy is of no concern, and thus the diffentiation between the above variants not required anyhow, the function float () as integer, the .int attribute or the math int () function converts any float to an integer; truncating for small numbers where the factional part is not zero, and otherwise creating an integer with at most 15 significant leading decimal digits, filled up with imprecise (decimal) zeros.

To convert an integer to a float, the .float attribute, the #= operator or the function math float () may be used. Only upto 15 significant decimal digits can be hold by a floating point number (see $SYSTEM.floatdigits), the rest is silently discarded as floats are approximations anyhow. If the floating point range (about 10^{6}) is exceeded, the result will most probably be *infinity*, but may also be a fatal runtime error.

The above function names have be selected similar to standard numeric functions like square root, sine etc, using

the math prefix to avoid one-word functions. For compact writing, the attributes will be used.

To support class function calls, in particular if libraries with other kinds of numbers are provided, aliases might be provided in the systematic form, despite the possible confusion between attributes and class functions:

```
integer: integer (@) abs
float: float (@) abs
rational: rational (@) abs
inteer: float (@) ceil
float: integer (@) float
integer: float (@) floor
float (@) frac
float (@) int
integer: float (@) round
float: float (@) sign
integer: integer (@) sign
rational: rational (@) sign
integer: float (@) trunc
integer: float (@) wide
```

As this may be resolved by providing field access functions allowing to define attributes or fields as functions, the above functions are probably not available.

None of these functions can be used to convert a string to a number; it is considered dangerous comfort, if a function tries to convert a string to an integer with the same function that converts a float to an integer; use e.g.

```
string (str) as integer:
string (str) as integer else (default):
```

If so required, write your own function:

```
int (mixed):
   ? is mixed float
      :> math round mixed
   ? is mixed integer
      :> mixed
   ? is mixed rational
      :> math round mixed
   :> string mixed as integer
```

The attributes for strings are:

name	function	Description
.count	count of ()	number of characters (code points, not bytes)
.IsRaw	is string () raw	
.IsAscii	is string () ascii	
.First	1	always integer 1
.Last	count of ()	Charcter count

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

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

As the character count of a string expression is seldomly required, the corresponding function is count of ().

The use of the same attribte .Count and function count of () for providing the number of characters as well as the number of elements of a bunch can lead to annoying errors if a function expecting a string is extended to accept a tuple of strings; however, this is not considered a common pitfall for newly written code.

Thus, providing .length or .size instead to avoid this minor trap was not further pursued; blocking another field name was not considered valuable.

Mutable items (rows, maps, chunks, fault items etc) have two common attribute:

name	function	description

.Serial	none	unique (small) number (order of creation)
.Revisions	none	count of changes since creation

Each creation of a bunch (row, map, etc) increments the serial number, provided by .Serial, which is intended as a short number to identify it uniquely when serialized.

The .Revisions attribute counts the number of modifications; thus it can be used to ensure consistency. It may be used in a scan to ensure the item is unchanged. As the count may be 16 bits only, i.e. provided modulo 2^16, only equality should be checked.

Bunches (tuples, maps or rows) have the following attributes:

name	function	description
.Cells	—	number of elements allocated
.Count	count ()	number of non-void elements

The .Cells attribute gives the number of allocated cells. It is the same as .Count for tuples and maps, and may be larger for rows if there are void cells; it may be larger than .Last - .First + 1. Note that .Bytes includes overhead, thus .Cells is not necessarily a factor of it.

The .Count attribute gives the number of elements in use:

- for tuples, it includes void cells (because normally there are none) and thus is .last.
- for rows, it is the number of non-void cells
- for maps, there are no void cells, so it is the number of keys used.

Rows have extra attributes:

name	function	description
.first	row () first key	key of first row element
.last	row () last key	key of last row element

As the .count attribute gives the number of non-void cells, it may be less than the number of elements actually allocated:

```
! r.count <= r.last -r.first + 1
```

There are two special control fields always attached rows or maps:

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

The .tag control field is used to identify (classes of) items, assigning a string to the control field .tag or as a parameter for the factory functions. It can only be set once: a fatal runtime error will occur if not void before.

The lock count is a means to protect a mutable item against changes. The .lock control field 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:

```
bunch () lock:-     \ increase the lock count and thus locks the item
bunch () unlock:-   \ 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 may be an unsigned short integer (max. 65'533 lock increments) and a fault item is returned if this maximum is reached. The runtime system can set the lock count to 65'535; any attempt to decrement it will return a fault item too, thus the bunch cannot be unlocked by the TAV-PL programmer. Note that locked bunches are not locked for deletion.

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

There is a standard hash function (system hash (x)) used for map access in the runtime system, that is quite quick and fairly good. It can be used for any item except a floating point number, as a hash function shall provide the same result for equal objects, and equality for floats is not supported here.

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

As a connector to external libraries, items of kind system are used. These are opaque objects that contain some state information to connect functions calls of the library, e.g. a file descriptor for file access. System items do not have fields, i.e. no user-defined information; if this is needed, the programme must use a bunch and put the system item in a field. Conceptually, system items are under control of the library binding code, and not intended to organize user data. Thus, there are no fields, only attributes and control fields; their names are defined by the binding code.

Implementation remark: ??? Some control field are stored like fields, with the colon prefixed, thus not accessible as fields.

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.

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

Integral numbers

Integral numbers are arbitrary precision numbers, normally not limited to the wordsize of the system that runs the programme.

In general, 64-bit words are used with machine arithmetic, even on 32-bit machines, as modern C well supports these.

Unless a multi-precision library is not available, numbers are automatically converted to and back, so there is no overflow in integer calculations. Conversion is done already at 60 bits to avoid frequent changes of representation.

As some operations and library functions reject the large multip-precision number, the function is () big may be used to determine the case. Also, the --sysinfo option shows how often the result was a big number and could not be converted back to machine integers.

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 future compilers can detect at compile time that more efficient code can be generated.

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.

Integer division in computers is not at all simple; 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 [7] gives a detailed comparison on this subject and demands that the following two conditions hold[8]:

```
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 TAV-PL, it is also desirable that the integer operators are consistent with the rational numbers, which will

be called the *quotients rule*:

```
n÷d = (n/d).trunc
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 *quotients 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 *quotients 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 *quotients rule*.
- Or follow the *quotients 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 TAV-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.

Note that commonly used (and perhaps the only one implemented) are the easy to remember `//` and `%%`.

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* (better use #E provided by the standard library). Note that the literal `3.1415` denotes an floating point number, although it is de facto the rational number `6283/2000`.

Thus, we have

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

However, the residue of a division of two integer numbers is not always the remainder or modulus of the division; this depends on a common divisor and the kind of division, see below.

Calculating with rational numbers benefits from simplifying (cancel common factors) 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 TAV-PL, 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   .trunc  .rsd   .frac  .sign
 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 TAV-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, or use the decimal floating point number that allows any precision. 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. Also, when using rationals, the denominators often grow very large, so it might be better to find a way to use (large) integers.

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* quotient. But as rationals can be intermixed with integers in arithmetic (as opposed to floats), comparison of accurate numbers will never surprise.

Rational numbers are currently limited to 32-bit integera as numerus and denominator; it is unlikely that they will be extended to arbitrary precision, although this would be fairly easy with the GNU MP library used anyhow.

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.

Normally, the hardware floating point is used by the C type double, which is nearly always a 64-bit word with 54-bit mantissa, according to the IEEE standard. See next section for extended precision floating point calculations.

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 (not for literals, unless in parenthesis)
- by the ##-operator
- by the library functions rational (x) as float or as float ()

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

The reasons for not even converting integers to floats automatically are:

- accuracy is impeded
- big integers may exceed floating exponent range
- the division operator / would require extensive rules, that the compiler will accurately apply, but the user

may err upon.

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

```
x =: ()
y =: ()
z =: x + y
! z = 0            \ result is integer 0
z =: ##x + y.float
! 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 is seldomly 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 x.float * y      \ ok
print (integer 3 float) * y    \ ok
print y * integer 3 as float       \ ok
print integer 3 as float * y       \ ok
```

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 .trunc 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.trunc + 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.trunc
? r > i.float
   i =+ 1
```

Rounding per .round is equivalent to:

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

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.trunc
! 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 .trunc and .sign return exact (integer) numbers, because it is easy to convert them to floating point, as in

```
! x = ##x.trunc  + x.frac
! x = x.trunc.float  + x.frac
! x = x.tunc + x.frac
! x = ##x.sign * x.abs
! x = x.sign.float * x.abs
! x = (float sign x) * abs x
```

Extended precision floating-point numbers

Commonly, floating point numbers are 64-bit long IEEE format with 53 bits binary mantissa (15 decimal digits) as supported by the hardware.

Extendend precison support might be implemented either in a library of its own, like the complex numbers (see below).

While arbitrary precision (big) integers are unlimited (except machine resources) and extended as required, this is not the case with floating-point numbers, which are rounded to a given precision by concept.

The underlying library alloctes container for a given precision, and the result of a calculation is always assigned to such a container with the precision defined for that container.

In contrast, TAV-PL does not use predefined typed variables, but instead uniform references to items.

By a plain assignment, just the reference is copied, not the number in its internal representation; thus here is not rounding.

If two items of the same precision are added, and both are accurate, the precision must be increased to the double limit, i.e. by one bit, unless rounding is used. However, floats are not accurate, because they are binary fractions possibly rounded in previous steps. If the precision differs, the smaller precision is used, as any additional precision will contain data that is unsure because the shorter operand was undefined there.

Subtraction is treated the same, with the argument that it is addition of the negative, leaving out the fact that differences of nearly equal numbers are normally significantly less accurate.

For multiplications, if the numbers are accurate, the sum of the precisions is required, as the result is just the product of the mantissas and the sum of the exponents. While standard floats round the product, this

Accordingly, any arithmetic with big floats and standard floats will result in a standard float.

The difference to big integers is that they are not automatically created from calclations with standard floating point numbers, but via a factory function:

```
mpf =: new float 256 bits       \ about 85 decimal places
! is mpf big
```

Note that due to the concept of floats, the

Decimal floating point library

Very high precision numbers are normally not required in practical uses, but especially in mathematical excercises.

Using one of the available extendend precision floating point libraries is possible, but with two drawbacks:

- internal representation is binary for efficiency; thus the fraction 0.1 cannot be represented accurate
- the precision is determined by the receiving variable, which is not practical here, where only references to items are passed.

From this reason, a library written in TAV-PL provides decimal floating point numbers using not fractions, but (big) integers.

The number 1.01 is represented as pair (101,-2), i.e. $101*10^{-2}$. Adding 0.00001 (represented (1,-5)) adjusts the exponent to the minimum, i.e. the first one to (101000, -5) the sum becomes (1010001, -5) ($101001*10^{-5}$). All numbers are stored normalised without trailing zeroes. While this sometimes shortens the numbers, rounding will nevertheless be necessary, see below.

Multiplication is simple, as it multiplies the integral parts and adds the exponents. This may result in rather large numbers unless rounding is used.

Addition and subtraction are correspondingly simple; just the larger exponent is reduced to the lower one, then the integral parts are added.

Division and taking the reciprocal requires rounding, as often the result is a (periodical) fraction, that would require infinite number of places.

For this purpose, the library sets a precision p used by the division. The integral part of the dividend is multiplied by 10^p, then divided by the integral part of the divisor with integer truncation. The exponent is the difference plus the precision.

The integral parts are normalized so that there are no trailing zeroes in the decimal representation, i.e. divided by 10 as long as possible.

Complex numbers

Complex numbers are not yet integrated in TAV, although this should be relatively simple: Just add a new kind complex, add the corresponding code to the operators and necessary or useful functions.

Instead, a library is available that uses pairs (2-tuples) of floats, and define a set of functions returing a (complex) pair, like:

```
complex (a) add (b):
   :> a.1 + b.1, a.2 + b.2
complex (a) re:
   :> a.1
complex (a) im:
   :> a.2
complex (a) length:
   :> float a.1*a.1 + a.2*a.2 sqrt
...
b =: 3.0, 4.4       \ parenthesis not necessary
a =: complex a add b
c =: complex  a add complex b mul a
print complex c as string
```

If instead of a tuple a bunch (map) tagged complex were used, class functions could be provided:

```
b =: new complex 3.0, 4.4       \ parenthesis not necessary
a =: a:add b
c =: a:add (b:mul a)            \ = a:add b: mul a
print c:as string
```

3.4. Strings

Strings are immutable sequences of characters, *character* meaning code point in Unicode terminology. Thus, there are more than 256 characters.

The number of characters in a string is practically unlimited, and the number of different characters 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, which does .Bytes).

To access individual bytes of a string is not supported; to process binary data, use chunks.

As Strings are immutable, any wording suggesting a modification must be read as *provide a copy of …*. In particular, the noun *clip* always denotes a copy of the respective part of the original string. While it would be possible to use a common buffer for the source and the clip, the added complexity is considered to expensive; any string function the returns a string either returns another reference to the original (whole) string, or to a copy.

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 (accents, '), the former transformed by message localization. There is no way to denote the encoding in TAV-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 functions or carving the result by string functions, see Formatting….

Void items are just ignored; items of other kind produce a fatal error.

Performance of string allocation and copy for long strings is often not relevant. Otherwise, collect the partial stings in a row and use the standard library function (that at some time may be written in host language) to join the parts.

Each character in a string can be accessed using row key syntax, starting at 1, and returning void if out of range or zero as key; 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. Relaxed attribute syntax for positive integer literals may also be implemented in future:

```
s =: 'abb'
! s.2 = s[3]
```

Negative keys 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]
```

```
    :> r
```

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

```
convert to integer (str):
convert to integer (str) base (base):
    \ if base n is zero or void, C language integer constant format is used
    \ returns fault item if string does not start with a number
convert to float (str):
    \ converts leading decimal floating point number
convert to string (n):
    \ accepts integers, rationals and floats
```

A lot of formatting can be done be using string functions, e.g.:

```
print pad right (sqrt 2) to 8 by '_'
```

See Formatting… for more capabilities.

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 often 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 (substrings) by pair or number range:

```
str =: 'ABCabd123'
st =: 4
! 'abc' = str[st, 3]
! 'abc' = str[st .. 6]
```

As a string is immutable, it can either be passed as a reference unchanged, or a copy of a part (including the whole) could be created as a new string item. Thus, using a slice notation as a target to modify a string is

impossible, and slice notation is only to retrieve a copy of a part of a string. This is a special — although important — case of a string function that provides a (new) string related to an input string object, of which are various according to various criteria.

Several of these scanning functions accept a starting point as parameter, as to avoid a copy of the string first, then the scan function:

```
string (src) count from (start) many of (accept):
    sl =: string src from start
    :> string sl count many of accept
```

Because many of these functions accecpt a starting point within the string, creating a partial copy is avoided, although less elegant. Example:

```
string (str) at (start) many of (accept):
    part =: str[start .. -1]
    :> string part many of accept
\ or
    x =: string str[start .. -1] many of accept
```

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.

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 keyed by code points between 0 and 255 in both cases, the encoder to raw will lookup values and return keys 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 TAV-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 UTF-8 encoding
\ see libraray documentation
```

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

String input and output

The standard input and output interface is line oriented, with each line provided as a string or supplied from a string. It uses the file streams defined by the C library, and includes pipes and command execution as well as network transmissions. It is buffered in the sense that the underlying input-output system may obtain and deliver data in parts of lines or as multiple lines; buffering tries to make the transfers as efficient as possible.

Furthermore, the transfers is synchronous, in that the program waits for the next line until it is filled and supplied, or until the line just provided has sufficient buffer space to be written.

As the standard line feed character is used as line delimiter, input and output is done in units of lines, that can be rather long strings if the file is a binary one. The line feed character is discarded.

If a line is either pure 7-Bit ASCII codes or valid UTF-8 encoded characters, it can be seamlessly be processed as strings. If a line is neither ASCII nor valid UTF-8, it is marked as RAW. Note that the lines may contain byte sequences that are valid UTF-8, but written in another coding. As the encoding is not recorded by the file system, it must be communicated otherwise.

In case a binary file (that does not consist of logical lines) must be processed or generated, direct (unbuffered) input and output to and from chunks of bytes should be used, which is also asynchronous, i.e. just provides the

bytes that are available or can be handled by the input-output system; so reads may be truncated and writes may tell that not all bytes have been written, and the rest must be transmitted in a new system call.

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.

Consider using the readline library, which allows line editing and does not echo the characters read, in particular not the trailing linefeed.

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.

TAV-PL Item Notation (TIN)

To save and restore maps and rows in text files, a notation is defined that uses ASCII-characters only and is assumend to be fairly good readable.

Notations like JSON https://en.wikipedia.org/wiki/JSON do not allow references to rows and maps might be applicable, there is no straight transformation of TAV-PL items, thus TIN is defined here.

White spaced is unified, i.e. any sequence of blanks, tabulators, new line and return characters is equivalent to a single blank, and may be shrinked and expanded without changing the net content. Thus, the text may be

rearranged break lines or join them on white space.

More than one item may be encoded, separated by semicolons.

Immutable items have a (short) representation by itself, which is repeated each time the item is referenced:

- void references use a pair of parenthesis without blank space between them
- integer numbers are given in decimal notation (with sign), using the underscore (not the apostrophe) for readablity, without blanks inside even for large numbers (must fit on a line).
- rationals are two integers separated by a slash, without blanks.
- real numbers use usual decimal fraction notation with a point as decimal separator independent of localization, and no blanks inside.
- booleans use the two symbols ?+ and ?-.
- strings are enclosed in either apostrophes () or quotes (").

In strings, code points (and byte values that are not code points) are represented using HTML entity notation; the ampersand & is replaced by &. Only the first blank in a row is encoded literally as a blank; any other following must use the encoding &sp; (or), as whitespace may be shrunk and expanded during transport. A second string literal that follows the first one after white space is catnated to it; this allows to split long strings to parts of e.g. 20 characters each.

Mutable items like rows, maps and chunks are referenced using an unique name, usually the serial number and revision count, both encoded with letters base 52, joine by an understore. As a reference, they are enlosed in the respective brackets ([], {}, <>).

They content is encoded once using the name, an equals sign, and a string enlosed in the respect

Numbers can be readable displayed with decimal digits in common notation. Chunks, being binary data anyhow, use base-64 encoding.

Strings are the most demanding items:

- String delimiters must be encoded
- Newline and tab characters must be encoded
- Blank space might be encoded optionally
- Control characters must be encoded
- Non-ASCII Unicode characters generally are encoded
- Non-Unicode bytes must be encoded
- Zero Bytes must be encoded.

For readability, the basis is the common string notation using quotes or accents as string delimiters.

All other characters (or bytes) can be represented using the HTML-entity notation, representing any ampersand (&) by &; see ??? literals.

Selecting the string delimiter might be:

- use quote (") only
- prefer apostrophes as there should be no further localization, but quotes if there apostrophes inside, but no quotes.
- use apostrophes, unless the string has the attribute.DoL10n, then use quotes.

A tuple is represented by a list of encodings, separated by commas, unless it is a pair, then in pair notation using >. If the elemente is another tuple, it is shown enclosed in parenthesis.

References to rows, maps and chunks use a unique name, composed of the serial number and revision count, separated by an underscore and both encoded by letters base 52. The name is enclosed in [], {} or <>. A proxy reference appends a tilde to the name.

The row or map itself is shown as the name, an equals sign, and a row or map literal (see…):

Example:

```
q_b = [ 'a', 'b', 3 ];
t_c = { 'haello' -> 'world''s end',
      'rpt' -> 3 };
```

Chunk are represented using base-64 encoding, grouping it as four characters to :

```
A_x = < UG9s eWbi B6d2 l0c2 NoZX JuZC Bhw59lbiBNw6R4
```

```
    Y2hlbnMgVsO2Z2VsIFLDvGJlbiwgSm9naHVydCB1bmQgUXVhcms= >
```

Fields can be added as extra lines:

```
q_b.numbers = 3;
t_c.src = "holal", 5;
```

3.5. Bunches (aggregates)

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

A bunch has:

- an arbitrary number of fields,
- either a row of cells holding arbitrary items, keyed by integer numbers,
- or a map of cells for items addressed by item references; often strings are used as keys.
- or called a bag, if it has only fields, no cells.

A row is also known as vector, column or array (of one dimension); while a map might also be called a dictionary, an associative array or a symbol table. The short words have been choosen for brevity of expression, although vector and dictionary would have been a bit more convincing.

Map keys can be any item, as they are compared for equality only, with the relaxation that items of different kind are just unequal and do not produce an assertion violation, see guarded equality, which is also used in tuple comparisons. comparisons. The exception are floating point numbers, that are always rejected as keys, as it is unclear when they are close enough to be *the same*, and could easily fill memory with map elements that were meant to be the same.

Integer numbers or strings are recommended as keys, and floats, if at all sensible, might be converted to strings.

Future versions might allow to define functions for comparison of classes, which will be used then.

A field… is addressed in the common dot notation that may also used for attributes, i.e. `bunch.field`. Row cells are addressed with the common key notation, i.e. `row[i]`, while maps use swiveled braces instead: `map{s}`.

To create a row or a map, either use factory functions or a pair of empty square brackets (`[]`) or braces (`{}`):

```
row =: new row
row =: []
map =: new map
map =: {}
```

Allocation of memory is done automatically as required; if memory is exhausted, it is a fatal runtime error. To avoid, use `new row (size)`. A row contains only references, thus the memory is proportional to the number of cells from the one with the lowest index upto the one with the highest index. Of course, if a row is filled with many very long strings, that memory is needed too, but a second row with the same data will not require the space again. For a map, memory is normally allocated only for the cells actually used (see details on maps).

Fields 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, map keys has to be used. (To clone bunches, library functions allow to get and set fields.)

Note that `r.x`, `r[x]`, `r{x}` and `r::x` are quite different: the first is the field with the name *x*, the second a row cell and the third a map cell keyed with the value of `x`, and the fourth call of a class function.

Fields allow to 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.

Similar to rows are tuples, that are immutable rows without fields.

In particular for tuples, constant integer keys can also be used in field notation, but not strings:

```
t =: 2, 3, 5, 7, 9, 11, 13
! t[5] = t.5
```

Rows

Rows are elsewhere called vectors, columns or arrays (of one dimension): A large number of arbitrary item

references can be stored and quickly accessed by an integer key (index). As the cells always contain item references, any kind of item can be used as data, and all kinds of items can be mixed. (Overlays — unions — to mix numbers, strings and other kinds are not necessary.)

There is always a smallest and a largest key used so far, they can be obtained by the attributes .First and .Last. There is no fixed origin of 0 or 1; negative integers can be freely used as keys.

Only machine integers (normally 64-bit wide) can be used to index a row, not arabitray long integer numbers.

A newly created row has .first=0, .last=-1 and .count=0. When the first cell is set, the integer used sets the fields .first and .last. By writing a non-void item with a key less than .first or greater than .last, the row is dynamically expanded as necessary.

All cells between .first and .last are allocated:

```
row =: []
row[-1000] =: -1000
row[2000] =: 3000
\ now 3000 cells have been allocated (at least)
! row.last - row.first = 3000
! row.count = 2          \ counts the non-void cells
```

The number of allocated cells is always .last - .first + 1 ; there is no attribute to calculate this number; .count is the number of non void cells. Note that the runtime normally allocates extra cells (at the top) in order to avoid frequent memory allocation if a row is filled with ascending keys; there is no attribute to tell how many cells are pre-allocated.

All cells are initally void, and so are cells outside the range of .First and .Last. There is no means to distinguish a cell with its initial contents from one which was overwritten with a void reference.

Howevert, for easier handling, .Last always tells the last non-void cell, i.e. if the cell at .Last becomes void, .Last is reduced automatically. No memory is released; just the unused cells at the end of the memory block can be reused by setting r[] or r??? r.Last+1.

Thus there is no *key out of range* error; just void will be returned on fetch, and the row expanded on storing a non-void item; storing a void outside is ignored and does not expand the row. Searching the next void cell in either direction never needs to check .first or .last, as a void cell will always be found.

Expanding a row may result in a fatal runtime error, if memory allocation fails. Note that several operating systems use optimistic memory allocation and never deny memory requests, but kill your process later if it dares to use all the allocated memory, and even swap space is exhausted. In any case, if your program uses huge amounts of memory intensively, swapping may slow it down, and make the whole system less responsive. Use the factory function new row size () to preallocate memory, which returns a fault item if not available, but there is still no guarantee for reliable operation due to the uncertaincies of the underlying memory management.

Rows do not shrink, i.e. memory space is never removed from the row, only set to void. To free the memory, a new row has to be made and the item references copied; note that this does not need extra memory for the data, as a row contains only references, and during the copy operation just the use counts are incremented and decremented.

A few library functions might help in standard situations:

```
row (r) copy            \ copy just the non-void cells, no fields
row (r) clone           \ full duplicate
row (r) append (rmt)      \ add at the end
row (r) shrink          \ copy without voids, new keys
row (r) from (f) upto (u)  \ copy part
```

To print the values of all cells including the void ones, use:

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

As often just the keys for non-void cells are required, there is a scan function to provide these:

```
r =: factory function to create and fill a row
?# i =: row r give keys
   ! r[i] ~= ()         \ void cells are not provided
   print r[i]
```

See below on Scan guards and scan functions for peculiarities if a row is changed inside a scan loop.

If the (estimated) number of cells is known at creation, a factory function allows to pre-allocte the desired number of cells:

```
r =: new row size 1000
```

If the number of cells cannot be allocated, a fault item is returned. The use of an integer literal within the brackets (e.g. r =: [100]) is deprecated.

Future versions may allow a tuple in the row creator to create a matrix, which currently requires the matrix library.

The notion r[] as an assignment target designates the next unused cell, i.e. r[r.last+1], even if there are void cells before.

To use a row as a stack, use the library routines:

```
push (stack) val (v):
   stack[stack.count+1] =: v
pop (stack):
   r =: stack[stack.count]
   stack[stack.count] =: ()
   :> r
```

Note that using a void reference as key results in a run time error, and does not automatically create a bunch.

Integers used as keys must not be big, i.e. their absolute value less $SYSTEM.intmax, which are at least 32-bit integers (with is () big not true). Rather large integers can be used; only the difference between the first and last must reflect the available memory.

Maps

In contrast to rows, maps are associative memories that can use any item not only as value, but also as a key for the cells. Maps are also called dictionaries, associative arrays or symbol tables.

Because they store the key as well as the value, they need more than twice as much memory per value than rows, unless key-value-pairs are stored in a row. Rows have an overhead of not-yet and no-longer used space, as they never shrink automatically.

Maps are created using an empty pair of swiveled braces (}) or the factory function new map:

```
map =: {}          \ digraph symbol
map =: new map     \ factory function
```

While often strings are used as keys, integers or tuples may also be used (where the integer 1 and the string are quite different):

```
map{1} =: 1
map{'1'} =: 2
map{9, "hello"} =: "Hello, 9"
! 1 = map{1}
! '1' ~= map{1}
! 3 = map.count;
```

Keys are compared for equality if their kinds match, otherwise are considered unequal (guarded equality); order comparison is not required. As floats intentionally cannot be compared for equality (only for almost equal), they are the only items excluded as keys. No callback for comparison is possible. To sensibly use floating point numbers as keys, the values must be rounded to some extent which heavily depends on the application. Using a string representation as key may help in this case.

Only those cells are allocated that have non-void contents; writing a void value removes the old key-value-pair. If the key does not exist, there is no error, just nothing changed, but the key is searched. In most cases this is more efficient than first checking the existance to avoid the error.

Thus, sparse vectors and matrices can use maps with integers or tuples as keys, as only the non-void cells are allocated.

In contrast to rows where a integer enumeration can be used to access all cells, the set of keys must be extraced by a library function supplying the keys unsorted (normally in last-used order):

```
r =: map m keys
?# k =: row r give values
   print m{k}
```

Often, scan functions are used to provide the keys or values:

```
?# k =: map m give keys
   print m{k}
\ or even
?# v =: map m give values
   print v
```

The scan functions save the set of keys (as references) when started. Thus the key scan delivers keys, even if the cell has been cleared, but no new keys added meanwhile are included. In analogy to the row scan, where void results are suppressed, the value scan does the same if the elements was removed since the start of the scan:

```
map (this) give values:
   mr =: map (this) keys
   ?# k =: row mr give keys
      ? this{k} ~= ()
         :> this{k}
```

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

Small maps use linked lists to store the values in the same way rows and maps store fields. Key-value pairs are unique; i.e. any value replaces the previous values for the same key.

When the number of elements increases, a hash table is used to speed up the search. Each slot still uses linked lists to finally obtain, change or add a value. The change is automatically as well as an enlargement of the hash table. In particular the latter distributes the re-hashing to following accesses, so there is no sudden delay when an element is inserted. The standard system hash function is used; it cannot be changed.

Library functions allow to obtain rows of keys, values and key-value-pairs :

```
row: map (@) keys:
   \ a row of all keys, unsorted
row: map (@) values:
   \ a row of all values, unsorted
row: map (@) pairs:
   \ a row of pairs of all existing keys and their values
```

Scan functions for maps save the keys in an internal row and then supply the keys from that row using the row scan:

```
map (@) give keys:
   ? $ = ()
      rok =: map @ keys as row
   |
      rok, $ =: $       \ restores old context
   \ $ is initially void, else the context of the previous call
   val =: row rok give values   \ the row values are the map keys
   \ end of scan?
   ? $ = ()
      :> val         \ last value, if any
   \ save row and nested context in this context
   $ =: rok, $
   :> val
```

The value scan as well as the pair scan all set aside the keys initally and then supply the actual values; thus changes in the map may result in void values, that are normally not possible.

To protect against this phenomenon, lock the map:

```
map m lock
?# v =: map m give values
   ! v ~= ()
   ...
map m unlock
```

Pre-allocating a given number of elements is not possible; if e.g. a map of large strings is used, the siginificant memory allocation is when the strings are created; the map entry itself is small, as only the reference is used.

A map requires between 50% and 100% memory overhead per key-value-pair. While only the references to the key-value pairs (plus a third word for the chaining link) are allocated and freed for reuse as required, currently the hash tables are not shrinked automatically. With an enlargment factor of 4, this would take place anyhow if the number of elements has gone below 1/8, ie. to 12.5%.

Users might it find error-prone and difficult that all kinds of items can be mixed as elements and keys; this can be restricted using assertions, in particular constraints.

Bags

A *bag* may be used if only fields, i.e. a map of compile-time keys, is required.

In contrast to a tuple, a bag is mutable, i.e. its fields may be changed, added and cleared after creation, while a tuple must be completely re-assembled if there is a change.

Like a row, a tuple requires space only for the item references, while bags needs two extra words per word item reference.

A bag has a tag and may thus be used for class functions, when the features of a map or row are not required.

Because the overhead for map compared to a bag is small, the former may be used until implemented.

Scans for rows, tuples and maps

Looping through a row can be done using a standard scan providing indices:

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

This will also supply keys for which the values are void; to provide — in analogy to maps — only keys for which the value is not void, a scan function is available:

```
?# i =: row r give keys
  print 'r[' _ i _ '] = ' _ r[i]
```

Using the loop context as last index, it can be written as:

```
row (r) give keys:
    ? $ =~ r.first - 1      \ if loop context is void
    ?* $ < r.last           \ search for next void
        $ =+ 1
        ? r[$] = ()         \ is void, try next
            ?^
        :> $                \ supply index
    $ =: ()
    :> ()
```

and similar:

```
row (r) give values:
    ? $ =~ r.first - 1      \ if loop context is void
    ?* $ < r.last           \ search for next void
        $ =+ 1
        ? r[$] = ()         \ is void, try next
            ?^
        :> r[$]             \ supply value
    $ =: ()
    :> ()
```

The row may be changed during the scan; if it is dynamically extended at the bottom (.first lower), these keys will not be supplied, while extending the row at the upper side will supply them. Values supplied will be actual values.

In order to keep the row unchanged, it may be locked (and unlocked) outside the loop, or the revision number checked inside:

```
row (r) give keys safe:
    ? $ = ()
        $ =: r.first -
        $.revno =: r.Revisions
        $.idx =: r.first
    ! r.Revisions = $.revno
    ?* $ < r.last           \ search for next void
        $ =+ 1
        ? r[$] = ()         \ is void, try next
            ?^
        :> $                \ supply index
    $ =: ()
    :> (
```

To obtain the set of currently used keys for a map requires a library function that return a row:

```
map (m) keys:-
```

Scan functions are also available for tuples for consistency, just using an integer range scan would be sufficient:

```
l =: 2, 3, 5, 7, 11, 13     \ primes
?# i =: tuple l give keys
   print i, l[i]
?# v =: tuple l give values
   print v
```

For tuples, rows and maps, a scan function is provided that just gives the values:

```
give (rt) values:
      key =: give rt keys
      :> r[key]          \ void gives void
```

Elements of a map 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 key 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[9].

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

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

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

As adding a row element to the next larger key is fairly often needed, an empty key, 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=* gives a fatal error. Note also, that for an empty row, the first key used is one, not row.last+1, which would be 0.

Proxy references (cyclic references)

As rows and maps may contain references to other rows and maps, cyclic reference chains can be created. This spoils the automatic storage deallocation, as such a chain is self-consistent, and might cause memory leaks.

A simple example to demonstrate a cyclic chain of references is a map that refers to itself in a field:

```
main (@):+
   a =: {}
   a.link =: a
   ! a.usecount = 2    \ variable a and field
   :> \ a memory leak will be reported
```

This can be the last element from a cyclic chain of elements where the programmer thought it easier to remove an element. The memory leak normally does not manifest, if during removal of a node the link in the node to be discarded is set to void, which should be used in defensive programming anyhow, as leaving it does not give any performance gain:

```
main (@):+
  a =: {}
  a.link =: a
  ! a.usecount = 2    \ variable a and field
  a.link =: ()
  :> \ a memory leak will be reported.
```

Other situations are a tree structure, where each node has a reference back to the root or the upper node.

As an aid to avoid such situations, volatile (weak or proxy) references can be used. A proxy reference is an item that contains only a reference to a bunch (row or map) without increasing the reference count. To avoid stale references, the runtime system sets this reference to void if the bunch referred to is destroyed.

A proxy reference is obtained by the unary operator `&`, and the original reference obtained by the operator `*`, which is rarely required. This is not a general pointer mechanism, as both operators do nothing if the operand is not a map or row (for `&`) or a proxy ref (for `*`) and just return the original item reference.

The above example with a proxy reference does not leave a memory leak:

```
main (@):
  a =: {}
  a.link =: &a
  ! a.usecount = 1    \ only variable a
  a =: ()             \ no memory leak
```

A proxy reference is a item by itself, and only one exists for each map or row; the operator `&` creates one only once. It has its own use count; if the corresponding map or row ceases to exist, just the reference inside is changed to void.

Because the only sensible operation is to obtain the reference, and in order to make usage easier, the item is automatically substituted by the stored reference.

The single exception is the function `is () proxy` which is the only means to distinguish between a proxy reference and its contents, including a test if the reference is void or equal to another reference. As there is at most one proxy for a map or row, checking the reference itself is the same as checking the references in the proxies. Assigning a proxy reference to another variable (or using it as parameter to a function) copies the reference to the proxy; and the copy of the reference shows the same behaviour.

There is one intentional peculiarity: A variable having a proxy reference may become void without explicitely assigned void:

```
r =: [ 1, 2, 3]     \ row created
x =: &r             \ proxy
! x ~= ()           \ a is not void
r =: ()             \ remove the row
! x = ()            \ a is void without changeing it
```

That's why the proxy reference is also coined volatile or weak.

The mentioned operator `*` obtains the (current) reference to the proxied map or row, and is thus no longer voilatile, in case this is required at any place.

In addition to the function `is () proxy`, the `error dump ()` and `error info ()` functions have an indicator for proxy items and just display the memory addess in order to avoid descending the cyclic structure expected. In this case, the dereference operator `*` might be useful to show the referenced item.

Although the items itself with a usecount greater 0 are known, this is not the case for the references to the items from outside, i.e. the local and global variables. Thus, a classical garbage collector is not possible.

The `debug dump ()` function also detects cyclic references and marks them while descending the references hold by bunches.

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 only the underscore and the point may be used in Returning bunches in scans.

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 =: 'org.tufpl.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 Returning bunches in scans, where the strings are more likely to be word lists e.g. SDL window.

For the programmer's convenience, the tag name may be given during creation with the factory function:

```
nm =: new map 'de glaschick tuf lang.env'
nr =: new row 'de_glaschick_tuf_lang_env'
```

As it makes no sense to localize the tags, do not use double quotes. Compilers may reject such an attempt. Of course, variables as well as named literals may be used.

System items, i.e. library bindings, may set strings other than SYSTEM, e.g. file stream, command stream like library bindings that use system items, in particular for class function use.

It may be possible to have tagged tuples to better support class functions, by providing a copy of a tuple with a tag:

```
tuple (t) with tag (tag):
    \ returns a copy with tag set
\ example:
nt =: tuple 1, 3, 5, 7, 9 with tag 'primes'
```

Setting a tag via attribute notation is not possible, as tuples are immutable.

The factory function might be exanded:

```
new tuple size (size) values (vals) tag (tag):
```

Sorting

The basic sorting operation is to re-arrange the elements of a row in ascending (or descending) order (inplace sort):

```
sort (row) ascending:
    ....
```

and with a user provided comparison function.

```
    sort (row) using `compare (a) and (b)`
...
compare (a) and (b):
    ? a < b
        :> 1
    ? a > b
        :> -1
    :> 0
```

When using a compare function, the sort is always *ascending* with respect to the return of the compare function; just invert the sign of return values to have it *descending*.

As only the item references are sorted in place, there are no extra memory demands, and the comparison of the keys may be the bottleneck. If a sorted copy is demanded, copy the row, then sort.

The standard library function uses insertion sort, which works fine upto 10000 elements, and is stable, i.e. does not change the order of already sorted elements (which might be important if a comparison function is used.) This sort is also used to provide a sorted scans for keys and values of maps.

To speed up sorting, the standard sort of the C library may be used, which is quicker, but not stable, and thus

denoted as unstable.

Often, a table with more than one column must be sorted by the first, and the others re-arranged correspondingly. In particular to supply the keys of a map in such an order that the values — not the keys — are sorted:

```
give keys of map (map) ordered by value
```

This can be done by three ways:

- each element is composite, e.g. a map with as many fields as columns, or a tuple with the same number of elements.
- sort two rows synchronously, i.e. apply all changes to the first equally to the second one
- supply a permutation tuple as the result of the sort

The first solution works by supplying an approprate comparison function, but is relatively inefficient.

The last one can be obtained by the second one by supplying an identity tuple (new tuple has (row.count) values ()

In the above case, at first two rows are created for the keys and the values of the map, and then sorted :

```
idx =: []
val =: []
?# i =: scan keys map
   idx[] =: i
   val[] =: map[i]
sort idx ascending sync val
?# i =: idx.scan
   print val[i] __ idx[i]
```

The corresponding insert sort reads:

```
sort row (row) syncing (val) ascending:
?# sp =: from row.first+1 upto row.last
   val =: row[sp]                        \ next value
   ?# ip =: from sp-1 downto row.first     \ go down sorted part
      ? val > row[ip]                 \ check in sorted part
         ?>                        \ insert point found
      row[ip+1] =: row[ip]               \ propagate upwards
      val[ip+1] =: val[ip]              \ propagate upwards
   row[ip+1] =: row                      \ insert here
   val[ip+1] =: val                     \ insert here
```

All sort functions return the row argument to allow nested use with functions returning a row, although most such applications are already provided, and it might never be used, except when a function reference is needed. Examples are not yet known.

Unions (storage overlays)

The C language has structures with static fields only, that are declared at compilation time.

Thus, it is often desirable to have an object with different sets of fields. These are unions of structures. Moreover, they are used if a function returns logically different objects, that are uninited by a single union.

In TAV-PL, this is neither possible nor necessary, as fields are dynamically used, so the union of all desired fields is arbitray large and needs not to be partitioned.

If a function delivers different objects, these are tagged and easily sorted out, in particular when dynamic class functions are used.

Within bindings of C libraries, this may be difficult to model.

For example, the SDL library (https://www.libsdl.org/) uses a very elaborate event union of about 30 unions.

Thus, the type of the event is used to convert the event to an empty map with the fields of the union set accordingly and an appropriate tag name.

It might be tempting to just save the struct and decode it on demand. But there is no gain in it: the event type must be converted to a string anyhow; and the code to set the map fields also. With dynamic fields, a small amount of cpu time might be saved at the expense of a complicated system, for which all others pay with 20% more for each item.

Rows and Maps compared to Python

In Python, there are four data types for collections of data:

- Lists: ordered, changeable, duplicate values
- Tuples: unchangeable lists
- Dictionary: collection of key value pairs
- Sets: unordered, unique values

Lists are very close to rows, as the index (key) is an integral number, thus ordered, and have duplicate values for any key. In contrast to Python, the lowest index is not always zero.

Tuples are named equally; a tuple is an ordered unchangeable row. Just that Python starts the indices (keys) at 0 and TAV at 1.

Dictionaries are very similar to maps, in particular upto Python 3.6 (or 3.7, depending on the source), because there is no insertion order maintained. Since 3.7, it is possible to obtain keys in the order in which they were inserted. Usage examples are not simple to find; one mentioned was the entries of a menu, for which a row of key-value pairs would be used.

One solution is to use a row of keys and a row of values. Finding a value means searching the key in the first row, and returning the value using the index for the row of values. Obviously, a library function to locate a value in a row and return its key is required and useful in other cases:

```
row (@) find (val):
    ?# k =: row @ give keys
        ? val = @[k]
            :> k
```

or

```
row (@) find (val) all:
    rv =: new row
    ?# k =: row @ give keys
        ? val = @[k]
            rv[] =: i
    :> rv               \ or :> row rv as tuple
```

With a generalised compare function:

```
row (@) find (val) using (func):
    ?# k =: row @ give keys
        ? func:( @[k] )( val )
            :> i
```

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. It is the programmer's obligation to ensure the binary data is properly interpreted, in particular number and encoding of strings.

Their primary purpose is to access binary files via the filesystem. Byte chunks are a variant of a bunch, i.e. support some attributes and arbitrary fields.

Byte chunks are buffers of a fixed sized, accessed and modified by standard library functions only. Once created, they exist until no longer referenced like any other item.

A byte chunk can is created using a standard library function, which returns a reference to the byte chunk, or a fault item if the space was not available:

```
br =: new chunk (n)
? is br fault               \ no memory
    fault br print
    br.status =+ 1
    :> bcr
! br.Bytes = n
```

The .Bytes and the .Count attributes return the number of bytes allocated. They are always greater than zero, because an empty chunk cannot be created.

A new chunk is virtually filled with zero bytes, either by doing so physically when created, or by tracking the fill level and zeroing on demand. In this latter case, the fill level is not available as an attribute.

To work with chunks, chunk slices are denoted by a triple

```
chunkref, offset, length
```

where offset and length must be non-negative machine integers. Negative offsets and lengths are assertion violations. Offsets are zero origin, thus 0 is the first byte. If the length is larger than the remaining buffer space, the minimum is used. If the length is zero or the offset beyond the buffer (i.e. offset > .Bytes), the slice is empty. Such a triple can be assigned to a variable for easier use.

The following function copies the bytes from the slice<src> (a triple as above) to replace the bytes of the slice <tgt>:

```
chunk <tgt> set slice <src>:
```

If the slices have different length, the minmum of both is used. The number of bytes transfered is returned, which may be zero if one of the slices is empty. If the slices refer to the same chunk, they may overlap and are treated as if the source is copied to a buffer, i.e. no propagation takes place. .

The standard library functions to access binary numbers from or to byte chunks are:

```
chunk <slice> get signed:          \ length = 1,2,4 or 8
chunk <slice> get unsigned:        \ length = 1,2,4 or 8
chunk <slice> get float:           \ length = 4 or 8
chunk <slice> put signed <i>:      \ length = 1,2,4 or 8
chunk <slice> put unsigned <n>:    \ length = 1,2,4 or 8
chunk <slice> put float <f>:       \ length = 4 or 8
```

signed and *unsigned* indicate integer numbers. If the length is not as given above, it is an assertion error, in particular empty slices. The functions return an item fetched (get) or t length (put). If the integer (signed or unsigned) does not fit into the number of bytes, a fault item is returned.

As an example, 7 32-bit signed integers are obtained starting at the 40th byte:

```
! is cr chunk
r =: new row
?# o =: from 40 times 7 step 4
   r[] =: chunk cr, i, 4 get signed
```

from (start) times (repeat) step (step):

```
? step > 0
   :> from start to start + step * (repeat -1) step step
```

A string of characters is transferred by:

```
chunk <slice> get string:
chunk <slice> put string <str>:
```

The first function copies creates a string from the slice given, which may be empty.

As the string may contain arbitrary bytes, including zero bytes, the result is always scanned before returned. Unless invalid UTF-8 byte sequences are detected, .IsRaw is clear and .Count adjusted to the number of code points. If the code points are all below 128, .IsASCII is set too. Otherwise, .IsRaw is set.

The second function copies the bytes of string<str> to the chunk slice. If the sizes are not equal (length ~= str.Count), the source is truncated or the target zero filled; this is different from chunk <> copy <> above.

To produce a zero-terminated string, use

```
chunk cr, offset, str.Bytes+1 put string str
```

To produce a string with a length byte, use:

```
chunk (cr, offset, 1) put unsigned str.Bytes
chunk (cr, offset+1, str.Bytes) put str
```

Parenthesis as shown are not necessary.

To obtain a string up to a delimiter, use the function:

```
chunk <slice> get string until <chr>:
```

The termination <until> can be an integer between 0 and 255, a tuple or a string; see the function's documentation

for details. The number of bytes used in the slice can be obtained by the .Bytes attribute of the result.

For convenience, two control fields are maintained by the above functions:

```
.Length        \ set to number of bytes last processed
.Offset        \ advanced by .Length
```

After creation, .Offset is zero and .Length the full bunch:

```
br =: new bunch 12345
! br.Offset = 0
! br.Length = 12345
```

If a pair is used instead of a triple, the second number is the length, and the offset taken from the control field. If only the bunch reference is used, offset and length are used from the control fields.

Byte order

As the binary data may originate from other machines, obtaining binary numbers depends on the byte order of the latter.

So chunks have a control field .Byteorder to indicate whether the byte with offset 0 is the least significant (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.

The control field .Byteorder may be:

- U for undefined
- L for LSB first (*little endian*)
- M for MSB first (*big endian*)

The function:

```
chunk <cr> byte order <new>:
```

always returns the byte order before the call as one of the characters U, L or M. If <new> is void, the byte order is not changed. Otherwise it must be string starting with one of the characters u, l or m, h or n in either case to set the byte order.

The letters starting may also be h for host, which is the byte order of the actual machine determined dynamically when used, or n for network, which is the same as M.

Anything else is an asssertion error.

Block IO

Byte chunks provide access to binary system files.

The functions to exchange byte chunk data with the file system are:

```
br =: new chunk <size>
fd =: new file chunk read <filename>
fd =: new file chunk create <filename>
fd =: new file chunk append <filename>
fd =: new file chung modify <filename>
rcnt =: file chunk <fd> read to <br, offset, length>
wcnt =: file chunk <fd> write from <br, offset, length>
loc =: file chunk <fd> tell position
lco =: file chunk <fd> postion <loc> start
lco =: file chunk <fd> postion <loc> current
lco =: file chunk <fd> postion <loc> end
rc =: file chunk <fd> close
```

The control fields .Offset and .Length are mained as above.

The following example shows how to copy a file in blocks (with minimal error handling):

```
infn =: "/dev/random"
outfn =: "/tmp/xx" _ system process own id
infd =: new file chunk read infn
outfd =: new file chunk outfn
bc =: new chunk size 1024*1024          \ 1 MiByte
```

```
?*
   / read as many bytes as are available until the buffer is full
   rcnt =: file chunk infd read bc, 0, bc.Bytes
   ? is rcnt fault
      :> 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 =: file chunk outfd write bc, 0, rcnt
   ? wcnt.IsFault
      :> wcnt
   ? wcnt ~= rcnt
      :> new fault -1 message "Write failed."
file chunk infd close
file chunk outfd close
```

A file might be opened as a chunk if the operating system supports memory-mapped files. To be specied and implemented; might be:

```
br =: new chunk maps <fd>
```

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 TAV-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 is 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 TAV-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, unless the fault item is marked as processed.

As fault items are not intended to be used instead of bunches, there is no distinction between fields and attributes. These field names may not be localised:

```
.checked    number of times 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.

To check if an item returned is a fault item, use the function `is () fault`. Normally, fault processing is done 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) set 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 — it returns void instead of a fault item — and thus it is not in the standard library:

```
ignore fault (item):
   ? is item fault
      item.checked =+ 1
      :> ()
   :> item
```

The `.backtrace` attribute, if not void, is a row representing the function backtrace (without parameters) at the point of error. Each element is a pair of source line number and function name, prefixed with the source file name.

To create a new fault item, a library function is provided:

```
new fault (errnum)
new fault (errnum) message (msg):
```

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
   ? is fd fault                \ 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
   ? is str fault
      :> str                  \ pass read error upstream
   ? str = ()                  \ end of file ?
      ?>                       \ yes, terminate loop
   print str
   n =+ 1
? is file stream x close fault
   :> 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:

```
file stream fd close
```

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 =: file stream x close
:> 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
   ? is fd fault                \ 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
      ? is str fault                  \ 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.backtrace attribute whenever a fault item is created, which is row of elements composed of pairs of line number and file name.

The pattern:

```
x =: do some function
? is x fault
  :> 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 assignment=? to indicate that a fault item may be returned, and in the latter case the function should return the fault item immediately upstream unchanged:

```
x =? do some function
! ~ is x fault
```

instead of

```
x =: do some function
? is x fault
  :> x
! is x fault
```

If the function returns void or a fault item, the target variable may be left out:

```
=? file stream x close
```

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 scans, the test is not so seldom delayed one statement.

In case that despite the arguments that follow, if 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 (errnum):
new fault (errnum) message (msg):
```

Fault modes for item operators

References to fault items can be passed around, as long as at least one copy of an unprocessed fault item remains; i.e. the use count is greater 0. Once an unprocessed fault item shall be discarded, the runtime system stops unconditionally with an error message and a backtrace.

Thus, it may occur that dangling fault items exist and make it difficult to determine the root cause.

Most functions allow only certain kinds or arguments, and might either pass the fault item upstream or stop using a assertion. In the latter case the dangling fault item stops the programme. Of course, a function with more than one parameter may receive two fault item simulaneously, and should use

```
! ~ is a fault & ~ is b fault
```

which will signal that two might have been received, but not any information about the fault items.

It has to be determined individually for functions how to react on exceptional cases:

- with an assertion that does an error stop
- with returning a fault item

This includes the option to pass a fault item received as a parameter, either unchanged or as a reference in a new fault item.

Operators that work on one or two items and deliver an item, can supply a fault item in case of an exceptional situation.

This is not possible for boolean operators and their expressions if the result is used to control the programme flow; this is a limitation of the fault item concept used here.

How item operators are dealing with fault items is set by the global fault mode:

1: stop

This is the inital mode. If an operator encounters an error (cannot deliver a decent result), it does not return a fault item, but it causes an error stop, including when a fault item is an argument. Operators that deliver void under certain conditions are unchanged.

2: delay

If an operator encounters an error, it returns a fault item unless one parameter is a fault item; in which case it causes an error stop. Prevents chains of dangling fault items in expressions with several operandes; at the same time allows to catch errors.

3: pass

If an operator encounters an error, it returns a fault item. In case one parameter is a fault item, it is marked processed and saved in the new fault item at .data.

To set or retrieve the fault mode, use the system fault mode () function that returns the fault mode when called; if the argument is not void and a string that starts with s, d or p, the fault mode is changed and should be restored later by passing the result of the call as parameter:

```
system fault mode (fm):
   mstr =: 'stop', 'delay', 'pass'
   ? fm = ()
      sfm =: $SYSTEM.faultmode    \\ or similar
      :> mstr[sfm]
   ? is fm string
      fm =: find fm in mstr
   ! fm >= 1 & fm <= mstr.count    \\ fails if fm is not a string
   $SYSTEM.faultmode =: fm
```

Future versions might define a subblock for a fault mode.

Note that the fault item mechanism is an effective 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 a map, the program will use up all memory, as with all other endless loops filling maps, 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 a scan?

Setting the scan context to void and returning the error item stops the iteration and leaves a fault item in the scan variable, which could be checked after the loop block, as the last value is still assigned to the scan variable.

Levaing the scan context other than void, will try to discard the fault item

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 scan 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 a scan 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 TAV-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 TAV 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 .backtrace 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.

The (keyboard) interrupt signal (*SIGINT*) is normally unchanged, i.e. terminates the programme silently.

In order catch the signal, a callback can be registered:

```
system signal interrupt `call back on interrrupt ()`
```

Calling this function set the function as interrupt handler; if the parameter is void, the interrupt handler is cleared and the signal reset to its default, i.e. terminating the programme.

The handler function is called with an error item as argument and may return either void or a fault item. If the return value is void, the programme is continued normally. A global variable may be set and polled later. If the return value is a fault item (maybe the parameter) which is injected to have any function return immediately with that fault item. This most often terminates the programme (depending on the fault mode), because either functions or operators do not accept a fault item, or the fault item is disposed while still marked unprocessed.

The callback is rather restricted and may, for example, not use file streams but set a global flag and return void. There is a special function to ask the user directly from the terminal, provided a controlling terminal (/dev/tty) is available:

```
terminal prompt (msg)
```

It writes (msg) directly to /dev/tty and reads a string in a way that conforms to the rules of signal service routines. If the process does not have a terminal attached, the result is void; if the user does not enter anything, the result is the empty string.

In case that a programme is in an endless loop, the keyboard interrrupt terminates the programme silently with return code 130, because the default signal terminates the programme. If debug is enabled (e.g. by --debug), and no signal handler defined, a short message and stack trace is shown, and the programme terminates with return code 126.

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 might be 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 key rows with string, or to add strings to numbers.

4. Assertions and Constraints

Assertions are statements that start with and exclamation mark (!), followed by logical expressions. They allow to declare and check invariants during runtime of the program, and by this means may help to stop faulty programs early that do not behave as predicted.

Constraints are named assertion expressions, that - 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 may be used in an assertion (as opposed to contraints), but are seldomly 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
```

As only expressions may be used, conditional expressions the shortcut *and* (&&) and *or* (||) are useful here:

```
! x = () || x > 0
```

Here, x can be void or an integer greater zero; note that void can be used in any comparison and yields false (unless compared to another void).

The shortcut evaluators are useful in cases like this:

```
! x.kind = 'integer' && x > 0 || x ~= ''
```

If a function accepts different kinds as parameters, 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 string  x
      ! x < '56'
      ....
      :>
   ? is integer x
      ! x < 56
      ...
      :>
```

By using a question mark, the following expression is printed (not yet implemented):

```
! a < 1 ? "Use integer instead of " _ a.kind
```

The message expression should be simple. Although functions may be called, other assertions might be violated.

Advanced compilers can support a block depending on an assertion; in this case, the assertion applies to all lines

in the block:

```
! 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 < 999` 995', 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).

Single-line assertions can be a pair, where the second part is evaluated and printed to the error stream, prefixed by *assertion violation:* or similar. The pair symbol is a bit misleading as the condition is not the violating case, but the opposite, so the `->` at this place must be read as :

```
! node.link ~= () -> "missing link", error dump node
```

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 compensate for the type system used in other languages. Constraint names have their own namespace; they are restricted to words (including the inner underscore that should be used to prefix library names etc.) to simplify statement parsing.

While assertions are executable statements inside a function body, constraints bind a name to an assertion expression to be applied on a single item.

The difference to a classic type system is not only that constraints are expressed flexibly, e.g. limiting the range of a number or the length of string, including a minimal length (which is probably very seldom used, but shows the flexibility).

Also, the classic type system needs a mechanism to override the restrictions (a*cast*) and even remove it completely. Assigning to a not constrained item is always possible and allows unrestricted processing of the item; if it is assigned back to another variable, any constraints for this variable must be honored.

Constraints are syntactically very similar to assertions, i.e. the line begins with an exclamation mark, followed the constraint name, a colon (or an exclamation mark) and an assertion expression, where the item to be constrained is represented by the scroll symbol @:

```
\ define 16 bit non-negative integer number
! n16: @ >= 0 & @ < 2^16
```

If the constraining expression yields true, the constraint is met; otherwise it is violated, in particular if evaluating the expression in normal code would give an error or fault. In the example, the first comparison implies an integer.

The clause *if integer, a natual number of at most 16 bit, else not constrained* would be:

```
! xn16: @.kind = "integer" && @ >=0 && @ < 2^16
```

A dependent block may be used if the colon is the last character:

```
! xn16:
  @.kind = "integer"
  @ >=0
  @ < 2^16
```

In this form, lines are checked in the order given, and must all yield true (as if connected by `&&`.

Complex conditions must be created by naming subexpressions and combine them by `&&` and `||`.

A variable is bound to a constraint by prepending the name and an exclamation mark, e.g. on a line of its own, before the variable is used the first time, similar to declarations in other programming languages:

```
n16! y
y =: x^2
```

The colon may not substitute the exclamation mark here, because a class function call may have been ment, and early compilers will not generate code at all:

```
x:find     \ must be x!find
x::find    \ call a class function
```

If an assignment changes a variable the (lexically) first time, the constraint can be prefixed

```
n16! y =: x^2
```

Constraining a variable before use is recommended in cases like the following:

```
do something (x):
   int16! y
   ? x = ()
      y =: 1
   |
      y =: x/2
```

Parameters and return values are constrained in the function template line:

```
n16! advance integer (n16! x):
   x =+ 1
   n16! y =: x / 2
   :> y
```

If the appearance looks clumsy, a colon may be used:

```
n16: advance integer (n16: x):
   x =+ 1
   n16: y =: x / 2
   :> y
```

Constraining the target variable can also be used in declarations:

```
! i32: @.abs < 2^31        \ integer implied
! n16: i32! @ && @ < 2^16
```

The use of the colon for binding the@ variable is not recommended if a single line is used.

Constraining elements and fields of rows and maps needs a constraint definition:

```
! lineRow:
   line: []              \ row of lines
! line:
   string: []            \ row of token strings, empty if comment
   string: .srcstr       \ the orignal source line string
   string: .fname        \ source filename
   integer: .lno         \ line number in source file
   integer: .indent      \ indentation count
   integer: .pos         \ current parse position
```

A tree might be declared like this:

```
! string: @.kind = 'string'
! tree:
   tree! .lhs
   tree! .rhs
   string! .op
```

Eventually, constraints may be declared cyclically.

As local variables cannot be nested by blocks, the binding of a constraint is valid for the whole body of a function and must the lexically first ocurrance either by the stand-alone version or as a target of an assignment. Note that the initial value of a variable is void and can only be changed by an assignment.

As the constaint definition uses a colon following the name, a colon may be used instead of an exclamation mark when binding a variable to a constraint, which might be considered better readable:

```
n16: do something (n16: x):
   t =: x + 1
   n16: y = t // 2
```

```
:> y
```

Constraints with an exclamation mark in the first column are global for the whole module (compilation unit).

A plus sign immediately following the exclamation mark in column 1 signals the constraint to be exported (by ^ or the *genhdr* utility).

Redefinition of a constraint is not supported. Future versions of the compiler might not reject it if they are equivalent, once the implementation is stable enough to specify equivalence.

Creating an alias definition is possible:

```
! alias: orig! @    \ or: orig:@
```

Constaint definitions local to a function are considered fairly useless and might never be implemented.

As global constraints are defined outside function bodies, neither local variables nor function parameters are possible; in contrast to assertions.

While named literals can be used — in particular for limits --, global variables are not allowed as to allow static evaluation.

To constrain a global variable, a very similar form is used, where the global variable name (including the dollar) follows the exclamation mark in colum 1:

```
! $myglob! n16
```

Although possible (because the name of the global variable, starting with a dollar sign, is not an allowed name for a constraing), using the colon behind it is not recommended.

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 may be determined already at compile time.

If a variable is not constrained, the compiler generates code to check it 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
```

Constraint verifications are checked only if a value is changed, 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: @[]
! mapOfIntRows: rowOfInts: @{}
! mapByStrings: @{string}
```

tell that the row has all integer values, the map only `rowOfInts` values, and that in `mapByStrings`, only strings are used as keys. Logically, the integer keys 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:

```
! intOrStr: integer! @ || string! @
```

Using tagged bunches allows to explain data structures using the *same tag* boolean operator `@=`:

```
! node: @.tag = 'node'
   @.lhs @= @      \ instead of @.lhs.tag = @.tag
   @.rhs @= @
```

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 differently, the last version is valid only in assertions and a shortcut to avoid cluttering of scrolls symbols:

```
@.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[10].

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[11]. 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 1000000'000'000'000' 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
   :> 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 ...
      :> 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[12].

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'
! fault: @.kind = 'fault'
! boolean: @.kind = 'boolean'
! integer: @.kind = 'integer'
! rational: @.kind = 'rational'
! float: @.kind = 'float'
! string: @.kind = 'string'
! row: @.kind = 'row'
! map: @.kind = 'map'
! tuple: @.kind = 'tuple'
```

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 a map.

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 a map 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.

The return values of a function may be declared using constraints:

```
float! square root of (float: x):
integer! square of (integer: i):
~mixed! @= 'integer' | @= 'float' | @= 'rational | @= 'string'
void! print (mixed! x):
```

Assertions for Bunches (rows and maps)

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 keyed. 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 `.lock` attribute to prevent accidental changes; however, the attributes are themselves not locked, so the bunches may still be changed.

maybe we need a mandatory constraint?

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`).

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:

```
r::append (val):
   @[] =: val
```

While variable aliases might be considered, just using variables is not a performance issue, but the use of aliases would be a source of error.

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[13].

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 one more character) 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, the parameters enclosed in parenthesis similar to a usual function definition, with more than one separated with words:

```
string (str) clip from (start) upto (end):
```

Because the parameter name is a single word, the closing parenthesis is not needed, but usually written. (Other special characters could be used to mark a parameter, but this has not been tried.) The enclosing parentheis have the advantage that a parameter can always be used in the function body this way, as extra parenthesis are redundant (but might clarify template scans, see below).

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; underscores (_) may be used inside words, i.e. not as first or last character. (Localisation may allow more characters for words.) Small and capital letters are fully distinguished. The first parameter may also be a single scroll (@) instead of a word, or a double colon (::) instead of a scoll in parenthesis triggering an advanced match process, see class functions.

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); two or more words are possible. 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.

Instead of the terminating colon, digraphs with a leading colon may be used at the end of the template line:

- 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. A block containing the statements for the function follows.
- If the last token is the colon-plus digraph (:+), the function is visible from outside the compilation unit. A block containing the statements for the function is expected following the template definition.
- If the last token is the colon-question digraph (:?), the function is visible from outside the compilation unit, but marked as deprecated, that should not be used further on, thus the question mark for *questionable*. A block containing the statements for the function follows.
- If the last token is the colon-dash digraph (:-), the template is a declaration for a function from another compilation unit to be linked, marked there as :+. Read as *minus body*, as no body statements follow. It is often in heaeder files, but may be used also directly. No body statements follow.
- If the last token is the colon-tilde digraph (:~), it is the same as :?, i.e. an external function that is deprected. No body statements follow.

There is no need to mark an internal function as deprecated.

The digraph := (as last token on a line) is reserved for function alias declarations, which are now obtained with a single line function body:

```
from (a) step (s) upto (e) :
   :> from a upto e step s
```

The performance degradaion is practically not existant; optimisations at other places are more fruitful, yet less easy to implement.

The program TAVPL_genhdr (or an option of the compiler) creates a header file from a normal source file. Each template ending in :+ and :? is copied, and the trailing digraph replaced by :- and :~. Additionally, the comment block immediately before is copied after the template. Also, literals for export (starting with ## in column 1) are copied to the header file.

The program tufh2html creates a HTML file out of it, that can be used for library documentation.

While not necessary, it is recommended to separate parameters by words, and not only by white space, otherwise use a tuple:

```
copy (a) to (b):    \ good
copy (a) (b):       \ avoid: one-word function, two parameters in a row
```

One-word only function templates without parameters are not possible as they cannot be syntactically distinguished from variables. If parameterless functions are desired, use two or more words (and not an underscore to join the words), as in:

```
next input:
   :> read from ...
x =: next input
```

One-word functions having a parameter should be avoided and restricted to a few very often used functions like print() due to their interfereces with variables:

```
print =: 5
print print nonl          \ = print (print nonl)
?# i =: from print upto 99  \ = from (print upto) 99
```

Often, they could be avoided by using a second short word in the function template, e.g.of, as, by etc.

See also ???naming

5.2. Function bodies

The body of the function is defined by the following (indented) lines, until a line 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
   x, y =: y, x %% y
:> 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 a commonly used exceptional 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. Nevertheless, setting variables first, even with void, is good programming technique and without any performance loss.

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.

It is conceeded that the parameter mechanism is effectively still a list of parameters, as in C, but wrapped in a different way.

5.3. Function call

A function is called if its template pattern matches, parenthesis are neighther required nor used (unless for expressions, see below):

```
sqrt (x):
   ...
r =: sqrt 2
move (a) to (b):
   ...
move 'aa' to s
```

Matching (part of) an input line is basically straightforward without backtracking, i.e. alternatives are not probed. This will in some cases undesirable, thus a pragma allows to switch a backtracking mode.

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 (a list of expressions). If one is found, it is noted as a parameter value, and the scan continues behind that expression and with the next position in the template.
- If the end of a template reached, the match has succeeded.

Future compilers might accecpt the end of the template at a parameter position and then supply void for that parameter.

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         \ = sqrt (2 + 5)
x =: sqrt 2 + sqrt 5    \ = sqrt (2 + sqrt 5)
y =: (sqrt 2) + sqrt 5  \ = (sqrt 2) + (sqrt 5)
y =: sqrt sqrt 16       \ = sqrt (sqrt 16)
```

Originally (and via a compiler option) expressions were possible only if a parameter was the last element of a template; within only terms were allowed, i.e. variables, array or map references and variables with attributes. Expressions had to be enclosed in parenthesis. But then a scan with a simple expression would be:

```
?# ip =: from (sp-1) downto row.first
```

The current solution is slightly more complex to understand, but needs far less parenthesis on average. For clarity, parameters may still be enclosed in parenthesis, as redundant parenthesis are simply ignored.

If there are templates that are the same until a parameter postion in one and a word in the other one, matching the word has precedence over using a variable with the same name; othewise, the second function would never be matched:

```
pr (x) nl:
   print 'not matched: ' __ x
pr nonl (x):
   print 'matched:' __ x
nonl =: 'nonl'
nl =: 'nl'
```

```
pr nonl nl      \ matches pr nonl (nl)
pr (nonl) nl    \ parenthesis needed
```

As shown, enclosing the parameter in parenthesis forces it an expression and thus matching the parameter position. The compiler will issue a warning message for the other line that a local variable is masked by the template matching; this message can — with the current compiler — only be avoided by renaming the local variable.

This has been mostly avoided in the standard library, except e.g. that a file name currently cannot be in a variable named stream, e.g. in the following line, the parenthesis are necessary:

```
r =: command (stream) lines
```

Because of this effect, the boolean functions that start withis use a less appealing template now:

```
is (x) integer          \ deprecated
is integer (x)          \ now
```

The compiler issues a warning if a template word is matched, and the same word has already been used as a variable. Depending on the situation, an error may follow, or the statement accepted, or behaves unexpectedly:

```
debug =: 1
is debug boolean    \ fails: expected 'enabled' instead of 'boolean'
is (debug) boolean  \ ok
is debug enabled    \ calls 'is debug enabled', not 'is () enabled'
is (debug) enabled  \ calls 'is () enabled'
```

Note that the third example might behave unexpectedly not calling the local function, thus the warning.

5.4. 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)
```

In contast to other statements, the expression may also be a boolean expression.

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 then replaced, the value is lost without notice (except unprocessed fault values).

5.5. Function references

Class functions should be preferred over dynamically selected 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.

A function reference is created by enclosing the template in backticks (*acute*, `):

```
funref =: `compare () and ()`
bunch.sort =: `sort record <a> less <b>`
sort map 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 that 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 one of the templates defined.

When calling a function reference, parameters are identified by position only; template matching is not available, because it is just the purpose to call functions with different and unknown templates.

To call the function given by a function reference, the digraph:((colon followed by opening parenthesis) is used:

```
funcref =: `float () sqrt`
x =: 3 + funcref:(2.0)
print x                    \ prints 5.1412
```

A tuple (list) is not the traditional list of parameters, but treated systematically as one parameter provided as tuple:

```
fr =: `max ()`
print fr:(1, 3, 77)        \ prints 77
```

To call a function with more parameters, the digraph)(is used to separate the parameters:

```
f =: `pad left () to ()`
print f:(33.4)(10)
f =: `pad right () to ()`
print f:(33.4)(10)
print f:(33.4, 10)         \ fatal: wrong number of parameters
```

Note that in an expression a closing parenthesis must always be followed an operator to continue, so there is no ambiguity.

As often callbacks may be void, if the function reference is void, nothing is done and void returned.

Now deprecated, library functions were used for upto 3 parameters:

```
call function (func) with (parm):-
call function (func) with (parm1) and (parm2):-
call function (func) with (parm1) and (parm2) and (parm3):-
```

Even if the functions templates may use angle brackets, the calling always uses normal parenthesis.

References to functions are immutable, but 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 and have the same number of parameters.

There is no need that the function is inside the same module, as just the entry address is saved.

The .count attribute is available for function references and returns the number of parameters. As the item is immutable, there are no fields.

One example for the use of function references 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
   ?# fr =: give funcs values              \ scan skips void
      ? 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 with 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 via system exit ().

5.6. Class Functions

To bind functions to bunches, in particular rows and maps, there are:

- field functions: a field can contain a reference to a function
- class functions: staticly connected functions to a class of items

In the first case, the assignment of a function to a field must be done individually each time a bunch is created. This allows maximum flexiblility, and thus maximum possiblities to do it wrong. Also, calling such a function does not support a template match.

The second case gives each item a characteristic name, which is used to select the proper function template. The characteristic name is provided by the .Class attribute (see [common attributes]) and is predefined for immutable items like strings and numbers, and can be set for bunches like maps and rows.

An example not using class functions might read:

```
my bunch (this) attr set (val):
   ! this.tag = 'my bunch'
   this.attr =: val
my bunch (this) attr get:
   ! this.tag = 'my bunch'
   :> this.attr
use the above templates:
   mb =: new map 'mybunch'
   my bunch mb attr set 5
   ! 5 = my bunch mb attr get
```

As the first parameter may be a single scroll, the above functions become:

```
my bunch (@) attr set (val):
   ! @.tag = 'my bunch'
   @.attr =: val
my bunch (@) attr get:
   ! @.tag = 'my bunch'
   :> @.attr
```

Using a double colon instead of the single scroll in parenthesis, the class name is better marked:

```
my bunch:: attr set (val):
   @.attr =: val
my bunch:: attr get:
   :> @.attr
```

The double colon :: (or scroll parameter (@)) in the function template delimits the class from the rest of the template, which is the core template. The class is a sequence of words delimited by blanks, see .Class attribute.

Within the body, the scroll symbol (@) denotes a parameter which refers to the actual (*this*) bunch, whether the double colon :: or scroll (@) is used as first parameter.

Calling a class function uses a double colon bigraph:: after a variable name:

```
x =: new map 'my bunch'
x::attr set 5
! x::attr get = 5
```

Instead of a variable, a field may also be used:

```
m.val::attr set (x::attr get)
```

Class functions for expressions are not supported, because:

- they undermine the constraint system
- assigning an item reference to a variable does not at all influence performance,
- would demand for an expression as statement,
- require often parenthesis, as the operator :: must have high precedence to bind normally to variables.

For example, the parenthesis in the following assignment would be required:

```
s =: (s::without trailing ' -') :: untab
```

```
\ normally in two lines, better understandable
s =: s::without trailing ' -'
s =: s::untab
```

As the latter two lines are a common pattern, where the function result replaces the original, because string functions cannot change strings in place, the right hand side of an expression may start with the double colon, in this case the target is substituted before, similar to increment and catenation assignments:

```
s =: ::without trailing ' -'
s =: ::untab
```

Class function calls may be used to deliver parameters in function calls in just the same way ordinary function calls may; Using parenthesis around a class function call is, if not necessary, often useful to make it better readable and to avoid mistakes. Better assign to a variable and use the latter, in particular for scan functions.

The template scan search collects all applicable class prefixes for the core template after the :: sign. If there is only one, the template is used as usual. (This may depend on the modules included.) Otherwise the compiler creates code that compares the class names to the class prefixes and calls the corresponding function if matched. If not matched, the error handling is determined by the fault mode.

As classes are created from maps or rows, all class functions for maps and rows are no longer available, once the class name tag is set; when required for the bunch itself, the explicit version must be used:

```
   new scene (ctx):
      rv =: new map 'scene'
      rv.ctx =: ctx
      :> rv
   scene (@) show:
      ?# i =: map @ give keys  \ NOT: @::give keys
         ....
Alternatively, a wrapper can be created for each of these to provide
the map function:
   scene (@) give keys:
      :> map @ give keys
```

The (outdated) class hashmap was an example where these functions are changed, and ahashmap was intended to be used instead of a map if required, with minor changes, if class function calls were used.

Howerver, as the class may not be changed once set, the inner map must be saved in a field, with the additional advantage that the map itself does not directly expose the map's content:

```
new scene (ctx):
   rv =: new map 'scene'
   rv.mem =: new map
   rv.ctx =: ctx
   :> rv
scene::add (x):
   mem =: @.mem
   mem{x} =: x
scene::show:
   mem =: @.mem
   ?# i=: mem :: give keys
      print i
```

Using class functions selection is slightliy quicker than a distribution function written in TAVPL (about 1%), but this difference must be considered irrelevant. Nevertheless, some of the early distribution functions have been removed to keep their number small.

See fault items for ways to catch the fault item, in particular the return value is discarded (unless it is an unprocessed fault item).

If the programmer, for reasons of efficiency or clarity, wants to bypass the selection at runtime, any function can be called always the long form with the leading class name.

A generic function for maps, e.g. row (@) give values is, on the other side, not per se a function for a new class:

```
st =: new stack
st::push 33
st::push 44
! 1 = st::find 33
```

Unless there is an alias declaration like

```
stack :: find (x):
```

```
    :> row @ find x
```

the last line will not find the ::find () function, as the stack has the class stack.

If a function can be used by two classes, it is necessary to define a template for each class, possibly as aliases, or as small wrapper functions.

This also applies for map and row functions in the standard library that can be used as class functions only for untagged maps. Note that the double colon :: or scroll (@) is required to tell the function is a class function.

There are no privileges needed to extend e.g. the string functions by own templates or define new mathematical functions or to create new number objects, e.g. complex numbers, and supply the common functions via the class function mechanism.

Some classes support the ::count function, which returns the logical number of entries, not the physical one of the top bunch representing the class, such that x.count and x::count produce different values. Providing ::count for basic maps and rows is nevertheless not recommended, as it spoils the subtle difference when applied to classes.

As with class inheritance, class functions may introduce subtle errors, when unexpected class functions are possible. For this reason, there is an option (currently the warnings option) that displays the classes if a class function is called and dynamic class selection is generated.

Instead of class functions, distribution functions determine dynmaically the function that finally does the work; they may be used if class functions are not to be used:

```
bunch (bunch) give values:
    ? bunch.Kind = 'row'
        :> row bunch give values
    ? bunch.Kind = 'map'
        :> map bunch give values
    ? bunch.Kind = 'tuple'
        :> tuple bunch give values
    :> ()       \ void, i.e. no-op
```

Blocked class functions

Class functions can be grouped in a block using the double colon starting with the class in column 1 and a trailing double colon (not yet implemented):

```
my class::
    attr get:
        :> @.attr
    attr set (val):
        @.attr =: val
    attr is (val):
        :> @.attr = val
new my class:                   \ single colon for plain function
    :> new map 'my class'
using the above:
    x =: new my class
    x::attr set 5
    ! x::attr is 5
```

An example using the SDL binding:

```
win =: new SDL window title 'Lorenz Attractor' rect rect  flags ()
rndr =: win::new renderer index () flags ()
rndr::clear 0, 0, 0
rndr::present
?*
    xi = (x * 10.0).trunc + #pixel
    yi = (y * 10.0).trunc + #pixel + #pixel//3
    rndr:: set draw color 0, 255, 0
    rndr:: draw point xi, yi
    rndr:: present
    \ instead of: SDL renderer rndr present
```

With short single word class names the advantage is debatable unless a block used as in:

```
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
```

In this case, the line ending in a double colon does not start at column 1, but at the usual indentation level, starting with a variable name followed by double colon: This allows the class check only once at the entry, and the compiler will complain if the variable is used as an assignment target within the block. Using it as a parameter is not affected, as this can only change data, but not the class (except the seldom cases that the class determining tag is not set in the factory function).

If the class name is longer, there is more gain:

```
rndr =: win:: new SDL renderer index () flags ()
rndr::set draw color 255, 155, 0
?# i =: from 1 upto 100
   rndr::draw point i*3, i*3
rndr::show
```

The constraint system may help to find out the class of a variable.

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
```

As the class function mechanism is not bound to a type system, the class of `org.cairographics` functions could be (locally) extended:

```
org.cairographics::print header (text):
   set font size 15
   select font face 'monospace'
   set source rgb 0, 0, 0
   move to 220, 40
   show text text
```

Within a class function, function matching starts with the class name prefix; if this fails, other templates are matched.

No overloading of class functions is possible; first, as it would make the code hard to understand, second, for easier implementaion.

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 =: @.ct
   ct::
      set font size 15
      select font face 'monospace'
      set source rgb 0, 0, 0
      move to 220, 40
      show text text
```

The user needs not to know the class for a library function, as it is derived from the template declaration (and so normally shown in the library documentation). Within the function body, it is `@.Class`.

Clearly only those functions are eligble for which the templates are included; otherwise a function reference must be used.

System functions, i.e. library functions may have class names

In a wrapper function like

```
bunch (rmt) give values:+
   ? is rmt row
      :> row rmt give values
   ? is rmt map
      :> map rmt give values
   ? is rmt tuple
      :> tuple rmt give values
```

```
   \ return void if none of the above
```

the first parameter is intentionally not (@) or ::, because it accepts as first parameter items with different classes, and is not intended for the non-existant class bunch.

If a function is to be defined for any number, all templates must be given:

```
integer (@) sqrt:
   ...
float (@) sqrt:
   ..
rational (@) sqrt (eps):
   \ needs precision (eps)
dfloat (@) sqrt:
   \ default precision if @ has none
```

Note the differing syntax:

```
f =: 1.141
print f.round      \ attribute
print f::round     \ class function syntax
print f:(7)        \ call the function referenced by 'f'
```

Note that attributes always return void if not defined, while (bunch) functions may return a fault item.

As tuples are lightweight, immutable rows, they have no tags and the .Tag attribute always returns tuple.

Using a single colon instead of the double colon does conflict with constraint declarations for the stand-alone declaration:

```
x:max        \ constraint or function call?
x!max        \ constraint
x::max       \ class function call
```

Inheritance

If just the class name were used, the class functions defined in the standard library for maps and rows would not be available without creating an alias. As there are no overlaps possible (setting the class name to a kind name is inhibited), the kind of the item (row or map) is also found. While this is a kind of inheritance, no other mechanism is used; if a class wants to use the class function of another class, it must define such a function:

```
class a :: function (parm):
   :> class b @ function (parm)
usage:
   x =: new class a
   x::function 1
```

Are class functions useful?

Without class functions, the programmer has either to know the full template name, and if required, write down the selection code himself. This is used in some cases in the standard library, in particular for common key and value scans, e.g. bunch () give keys or give () keys.

However, this way is fairly limited; if class functions for ::as string would be provided, more items could could be printed.

This requires that when there is no class match, a fault item is returned, so that the print function can print a replacement.

The disadvantage is that of general inheritance: it is hard to determine for the programmer which function is eligible at that place. The compiler knows well, but the user can err. Thus, the compiler should have an option to en- or disable class function calls, or issue a mesage giving the known classes at the point of call with the libraries visible there.

Constructors and destructors

The normal way to create an object is to use a factory funtion, which play the role of constructors. Because a variable can hold a reference to any item, there is no information on which kind to create.

Normal automatic memory deallocation is nearly always sufficient.

There are two possiblities for destructor functions:

- a function is dynamcally connected to a bunch, normally by the factory function.
- a fixed class function is called for every deallocation.

For the latter, no satisfactory solution has been found so far.

For the first case, the control attribute .OnDelete can be set with a function reference with a single parameter to be called, which is the reference to the item that is finally deleted when the function returns. Any returned value is silently discarded.

Once called, the attribute is disabled to prevent endless cycles.

The function might e.g. clear cyclic references, unsubcribe from a service etc.

This feature does not slow down normal bunch deletion.

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 size (n):
   rv =: new row 'queue' size n
   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 key 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 1 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 size 100         \ factory function
   ! q::dequeue = ()
   q::enqueue 5
   q::enqueue 'hallo'
   ! q::dequeue 5
   q::enqueue 3.5

condensed:
   q =: new queue size 100         \ factory function
   q::
      ! dequeue = ()
      enqueue 5
      enqueue 'hallo'
      ! dequeue 5                   f
```

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 a map
- The constructor may not provide enough options
- During initialisation, the state may be incoherent

5.7. Export and import of functions

Functions templates ending in a single colon 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 :+
   ...
```

To use a function exported from another module, just the template is given, ending in :-, in which case no body may be supplied. This could be regarded as an import declaration, and the compiler may allow them several times, if the templates are the same.

It is used to define the prototypes for library functions, without appending the source code (which is impossible for some basic library functions anyhow). There is — for clarity — no overloading (masking out) of imported library function templates. Thus, all external functions of a library (and also of the libraries used) cannot be redefined.

This means that extending an existing library by new functions might break existing code, and in such a case, a new version with a different library name shall be created, which is good engineering practice anyhow. Of course, using the object oriented mechanism of class functions allows to define new functions with a significantly smaller collision probability, even if it is still possible, as class functions just remove writing effort in composing function templates.

There is a set of functions required by the runtime system, plus a lot of commonly used functions defined in TAVPL_baselib.tufh, namely the functions in the runtime, string functions, functions for file acccess and mathematical functions. Some often used modules, e.g. crypto,, and for multi-dimensional operations are added for the collection TAVPL_stdlib.tufh. More seldomly used libraries, likereadline and XML parsing, are kept separat and must be included individually.

The modularisation for TAVPL_stdlib.tufh and TAVPL_baselib.tufh may change in near future, if the structure is reorganized.

The compiler might accept :~ instead of :- for functions that are deprecated and issue a warning once the function is used. The convention to mark deprecated functions for the header generation currently is the comment starter \(~.

Note that deprecated attributes are currently determined by a list in the compiler source.

5.8. Function aliases

As the linker alias feature can only be used if the parameter list is the same, function aliases are always written as return-only bodies:

```
clip (s) upto (i):
   :> string (s) clip upto (i)
is character (c) in string (s):
   :> has string (s) character (c)
```

Calling a function is the most efficient operation in TAV-PL, as it is always compiled directly to the host language C, so there is no measurable perfomance penalty. (Unless the compiler detects the case — similar to tail recursions — and generates appropriate linker instructions, if possible)

The digraph := instead of : at the end of the template definition isreserved for the purpose of alias declarations, whatever the syntax may be.

5.9. Naming conventions for function templates

In particular for libraries, a certain structure in the function templates is useful. Local functions that do not conflict with other modules or libraries should benefit from the fact that the templates allow a natural wording:

```
calculate checksum for (fn) using (sumpgm):
```

In this manual, the wording and structure is used for English.

In general, the template should start with one or a few words telling the subject, i.e. an object oriented approach:

```
file stream (fd) flush
row (row) sort ascending
string (str) locate (needle)
```

Systematically prefixing also parameters with a word describing the kind could be tempting:

```
string (str) locate string (needle)
string (str) times integer (ni)
```

However, this leads to duplicate words, if the parameters are function calls, as in:

```
n =: string tgt locate string string 'a' times 5
n =: string tgt locate string 'a' times 5
```

Also, when the constraint system is used, these indicaters are not needed.

Thus, only the leading word (or words) — often a noun — is used to indicate the domain of functions, in particular the kind of the first parameter.

If the result is boolean, then the template contains the word is. If it starts with is, the parameter to be enquired follows, and then words to tell the nature of the check. These functions (should) never fail, regardless of the parameter's kind, just return void if not appropriate:

```
is (x) float
```

Otherwise, is or has is the word after the first parameter:

```
string (str) is ascii
```

Words are often used to indicate the following parameter:

```
using (func)    function reference
with (parm)     parameter
size (int)      number of elements
length (int)    amount
by (x)          separator
from (int)      moving start point
at (int)        fixed start point
upto (int)      end point forwards
downto (int)    end point backwards
to (target)     any target
```

If the result is of a different kind, the word *as* is used, in the sense of *convert to*:

```
string (str) as integer:
string (str) as float:
integer (int) as string:
float (flt) as string:
rational (rat) as string:
boolean (b) as string:
```

with more variants, e.g. indicating the base:

```
   string (str) as integer base (base):
   string (str) as integer else (def):
Some functions have aliases, in particular if the function
has only one parameter:
   as integer string (string):    \ string (string) as integer
```

The item kind word might be dropped for brevity:

```
as integer ()
as float ()
as string ()
as row ()
as map ()
as tuple ()
```

These functions are often generic, where the kind of the argument is dynamically determined:

```
as string (x):
   ? is x integer:
      :> integer x as string
   ? is x float:
      :> float x as string
   ...
```

Scan functions use the word give:

```
row (@) give keys:
map (@) give keys:
```

If the function is generic, they start with give and have the target at end:

```
give keys (bunch):
give values (bunch):
give values (bunch) ascending:
```

Generic functions accept different item kinds as primary parameter and internally select the appropriate function, or return void. They use item as prefix; which item kinds are useful, depends on the implementation of the function:

```
item (any) count:      \ .Count attribute
item (any) hash:       \ hash value
```

Generic scan function start with the word give; see the specific function's documentation of what any can be.

```
give (any) keys:
give (any) values:
give (any) keys ascending:
give (any) values ascending:
```

They are like restricted class functions, as they cannot be extended by providing more templates. If the class is known at that place, the specific class function should be preferred.

To avoid confusion, the words from, upto, donwto and to are normally used for ranges, and take, draw and as etc. used for changes.

Some aliases drop as row or as map:

```
command (cmd) lines
map (map) keys         \ instead of: map (map) keys as row
```

Factory functions use the word new prepending the class name:

```
cs =: new command stream reads "file"
ln =: command stream cs get
```

Using class functions when implemented, the tail of a template can be attached to a variable:

```
m =: mew matrix sized 3, 5          \ tags the bunch with 'matrix'
?# tup =: m::give keys              \ calls matrix (m) give keys
   m[tup] =: tup
m:print                        \ instead of map (m) print
```

As the compiler will anyhow try to find a template that has after the first parameter the tail locate () etc. as in the following lines:

```
p =: str::locate ';'
wrd =: str::from 1 upto p-1     \ calls string  (@) from (f) upto (t)
i =: str::as integer
```

Mathematical number attributes are provided as functions using the math prefix:

```
math sin (x)        \ not: math (x) sin
```

However, the corresponding class functions will be:

```
float (@) sin:
   :> math sin @
pi =: #Pi
print pi::sin, #Pi::cos
```

Except the number generators, scan functions are characterised by the word give:

```
?# i =: row r give keys
   ...
```

One-word functions (with a parameter) should be avoided, in particular in the standard library, as their word interferes with normal variables:

```
print =: 5
```

```
print print nonl    \Bad: = print (print nonl)
print (print) nonl  \ ok
```

For this reason, only small number of one-word functions are defined in the standard library, and in particular the standard mathematical functions are prefixed by math.

Commonly used functions provided as one-word functions are:

```
print ()       \ by itself
join ()        \ row (@) join ... ; map (@) join ...
quote ()        \ string (@) quote ...
```

The join and quote

Not available as one-word functions are min and max; they are not so frequently used to justify one-word over two word functions:

```
max ()         \ use 'max of ()' or 'maximum of ()'
min ()         \ use 'min of ()' or 'minimum of ()'
```

There are aliases for tuples, rows and maps and thus can be used as class functions:

```
tuple (@) max:
   :> maximum of @
tuple (@) min:
   :> minimum of @
x =: 3, 2, 5, 4
print x::max, x::min       \ prints 5 2
print (3, 2, 5, 4)::max    \ possible, but useless
```

If the argument is not a tuple, row or map, it is returned unchanged.

Providing them as attributes was discarded, in particular as it is expected that attributes are quick and might not take a long time, which would be the case for maps and rows in particular.

Mathematical functions in general use the math prefix. The simple number conversions can be obtained via an attribute or function (e.g. .Abs or math abs ()). They work exactly like the attribute, i.e. deliver void if not applicable.

All other mathematical functions use the same prefix, but are mostly callable as class functions.

The ::as float conversion is the same as using the## operator, and often requires parenthesis anyhow:

```
! (float 3 + 5) = ## (3 + 5)
! integer 1 + 2 as float > rational 14 / 5 as float
```

The .Count attribute is heavily used, but seldomly required for an expression, thus a two-word function is provided:

```
a[i].count     \ count of a[i]
count of a[i]  \ use function
```

For formatting numbers, see Formatting….

6. Standard library

6.1. File I/O

There are currently two sets of library functions:

- to read and write text files (strings via streams)
- to read and write binary files (chunks)

Often, text files are sufficient; this interface consideres the file separated into lines by linefeed characters. Lines are often obtained by a scan function; or the whole file is read into a row. If lines are written, the linefeed character is automatically appended to each string that is issued over this interface. As strings can be very long (upto 2 Gigabytes), even texts without line feed characters may be processed this way. However, zero bytes are not possible; which is a limitation of the underlying streams library, not one of TAV-PL strings. Furthermore, the characters are preferrably encoded as UTF-8, which includes pure ASCII, both without zero bytes. Other encodings are possible, but not easy to handle.

When input is a terminal, input is considered a line even without a linefeed character, if the end of a line is

determined by other means.

To access binary files, ???chunk must be used.

Similar to the text file interface, pipes can be used to send lines to a command or receive lines from a command, with the same condition as for files.

6.2. Network I/O

Basic I/O

The basic network interface is similar to the access of text files — Lines of text — terminated by linefeed characters — can be sent and received, either per TCP or UDP. The library handles long strings in a reasonable way and may deliver strings as lines even if there is no explicit linefeed character. As with files, every string sent will be converted to a line, usually by adding a line feed character; and on receive, partition the input into lines, in particular if the separated by line feed characters.

As strings may contain zero characters, and the used interface transparently passes zero bytes, the sent and received lines may contains such beasts.

Note that the basic I/O is suitable for a HTTP library programmed in TAV-PL, even if some string functions are system library functions for efficiency.

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 later.

6.3. Formatting

Formatting with functions

Composing a print line is fairly easy without the complexity of a format pattern:.

```
lne =: "Results: "
lne =_ m.res
lne =_ " median "
lne =_ median of m using 33
print lne
\ or \
print "Results: " _ m.res _ " median " _ median of m using 33
\ or (using a list) \
print "Results:", m.res, "median", median of m using 33
```

A field width can be obtained by a padding function (the padding operators_> and _< are not yet implemented):

```
print a, pad left i to 10, pad right j to 20, pad right k to 15 by '.'
print a, (pad left 10 i), (pad right 20 j), pad right k to 15 by '.'
print a, 10 _< i, 20 _> j, pad right k to 15 by '.'
```

The count of decimal digits for (non-negative) integers is easily determined if converted to a string:

```
x =: as string 12354
! x.count = 5
print x.count, x
print (count as string x), x
print (count x_"), x
\ same provided by library function for any base:
integer (arg) count digits (base):
```

Note that there is no need to allocate buffers in advance to any conversion, thus there is seldomly a need to determine the number of characters in advance.

To convert an integer from and to any base, library functions are available:

```
integer (i) as string base (basespec)       \ does not prefix Ox
string (str) as integer
string (str) as integer skip (chars)
string (str) as integer base (base) skip (chars)
string (str) as integer base (basespec)
string (str) as integer clib (base)
```

The last function accepts all the variants that the c library allows, e.g leading zero for octal, leading0x for hexadecimal, etc.

The other two not only accept a number as base, but also a string for any characters as digits.

The skip variant allows to ignore certain characters, in particular grouping characters.

To produce such a format, use string catenation:

```
print '0x' _ integer i as string base 8
```

Grouping

In particular for reading large integer numbers, it is sometimes useful if the digits are grouped when printed. Because this impedes the re-use of the numbers, it is not use per default.

A library function groups numbers:

```
\( insert (gc) every forth decimal digit counted from the end;
   reset count with each non-digit
\)
string (str) group (gm):
    ...
\ useful aliases
string (str) group:
    :> string str group ()
group (str) by (gm):
    :> string str group gm
group (str):
    :> string str group ()
usage:
    x =: 1234567
    print group x                \ prints 1'234'567
    print group x by ' '         \ prints 1 234 567
    print group x by ' ', 2      \ prints 1 23 45 67
```

Grouping does not change an already grouped string, as the group character is inserted when the fourth digit in a row (from the end) is encountered.

The parameter can also be a pair of the group character and the number of digits, which defaults to 3.

If more than one number has to be grouped, parenthesis are required: (except the last one):

```
x =: 1234567
print (group x), group y
print group x, group y          \ print group (x, group y)
```

This works as intended, as the group function groups each element of a tuple and returns the result as tuple, which is possible because grouping does not change an alredy gouped string.

To use class functions as in:

```
x =: 1234567
y =: x.float + 0.2
Z =: x / 33
print x::group, y::group, z::group
```

requires additional aliases:

```
integer (@) group:
    :> group @
float (@) group:
    :> group @
rational (@) group:
    :> group @
```

The default group character and number can not be changed, because retrieving it from a globally changeable

variable undermines program reliability.

Numbers converted to strings, e.g. by catenation or as parameter for print, are not automatically grouped. Not only would this spoil existing code, smaller numbers do not really benefit from grouping. The format (fmt) (list) function has a flag to do gouping.

Floating-Point Numbers

The commonly used 64-bit binary floating point hardware can convert a number of upto 15 digits back to decimal; thus, it is a good rule that more digits are normally unreliable. This magic number of 15 decimal digits is used verbatim from now on, as it is improbable that any alternative will be used extensively.

For floating point numbers, formatting is more demanding, as the maximum precision of 16 digits is often senseless, and to save space and increase readablility, the scientific exponent notation (standard form) is necessary in addition to the exponent-less decimal form. Both require that the number of decimal digits shown can be controlled.

The factional part is often called *mantissa* if between 0 and 1, and the *significand* if between 1 and 10, but this terminology may not be followed strictly. In exponential format, the number of digits after the decimal point of the significand is called the *precision*.

In order to clearly mark strings derived from floating point numbers, the always contain a decimal point (subject to localisation). While the tradition to allow a decimal number to start or end with a decimal point has its merits, because it sometimes avoids insignificant zeroes (i.e. .5, 3.), it is prefered to have the decimal point always surrounded by digits.

Even if the number of non-zero digits may be limited (to avoid insignificant digits), no insignificant zeros are produced except a single zero before the decimal point and a chain of zeros after it (followed by a non-zero digit) which are commonly known as means to set the magnitude of a number.

A single zero after the decimal point is significant unless the number of digits before is greater than 14. To avoid this case, the conversion functions switch to exponential format it the trailing zero after the decimal point would be an insignificant one.

To write an uncommon conversion, the following functions may be used:

```
float (x) split decimal:
   \ returns a pair of integers (i, e) such that x = i * 10^e
float (x) split binary:
   \ returns a pair of integers (i, e) such that x = i * 2^e
```

Standard (General) Format

The standard output format is generated by:

```
float (f) as string:    \ shortcut for: float (f) as string ()
as string (f):
f::as string
f _ "                   \ catenation with the empty string
```

The standard format has

- upto six significant digits,
- always a decimal point with a digit to the left and one to the right
- removes trailing zeroes, but retains one
- uses exponential format if insignificant zeroes were required

The latter is the case if more than two insiginificant zeroes were required after the decimal point, i.e. the argument is less 0.001, or there are more than five digits before, i.e. the argument is greater than 99999.9.

Examples:

```
1.23400e-5      \ too many insignificant zeroes after the decimal point
0.001234
0.01234
0.1234
1.234
12.34
123.4
1234.0
```

```
12340.0
1.23400e5       \ no single tailing insignificant zero
```

As could be seen, the number of digits shown is at most 6, but may be less by removing trailing (significant) zeroes.

To allow a different number of digits, the general conversion function allowing to select the number of signifcant digits is:

```
float (@) as string (d):
   ....
float (@) as string:
   :> float @ as string ()
```

The number of digits in the result is controlled by the parameter `(d)`:

- If void, the above scheme for string catenation is used with upto 6 digits
- If `0.001 ≤ |i| ≤ 1.0`, an insiginificant zero, a decimal point, upto 2 insignificant zeroes and `d` decimal digits are produced.
- If `1.0 ≤ |i| < 10^(d-1)`, a decimal point is inserted at the proper positon between the `d` decimal digits.
- Otherwise, exponential format is used.

No blank space is generated; the number of characters generated is between `p+1` and `p+7`; use the function `pad...` to provide leading blanks. To have a plus sign for positive numbers, use

```
float (@) as string (d) signed:
   ? @ < 0
      :> float @ as string d
   :> '+' _ float @ as string d
```

No error may occur (except not a float item).

Universal (Exponential) Format

The universal (decimal) exponential format allows a given precision which is the number of decimal digits after the decimal point, so the result has `d+1` decimal digits in the significand (d is limited to 1 upto 14):

```
float () as string digits (p)   \ exponential, d digits precision
\ produce exponential format like -1.23456e00 if p=5
```

As in the C printf format, the exponent has two digits at least for better readbility, and the is `d+6` upto `d+9` long; at most 25 characters. No blank space is generated; again, there is a simpl function to have a leading plus always:

```
float (@) as string digits (d) signed:
   ? @ < 0
      :> float @ as string digits d
   :> '+' _ float @ as string digits d
```

To change the exponent character, e.g. `₁₀`, use

```
r =: float x as string digits 5
r =: string r replace first 'e' by '₁₀'
```

Engineering applications might require the exponent as a multiple of three, so there might be implemented later:

```
float () as string digits (p) step (s)
```

Fixed Point Format

To display a floating point number with the decimal point at a fixed position from the end, the following function accepts such a count between 1 and 14 as precision, similar to the exponential format:

```
float (x) as string fixed (p)   \ fixed with precision
\ produce fixed format like 123.450 if p=3

   0.001
   0.012
   0.123
   1.234
  12.345
 123.450
1234.500
12345.600
```

Because only the number of digits after the decimal point is fixed, the maximum number of significant digits before the decimal point is 15-d. If this would be the case, i.e. x ≥ 10^(15-d), exponential format with all 15 digits is used, because otherwise insignificant and excessively many useless digits would result (as does the f format in C)-

There is a variant

```
float (f) as string strict (d):
\ returns fault instead of exponential format
```

that returns a fault item (with the exponential string in .info) instead of the exponential format, to aid reliable programming.

Constant field width

Another, rather uncommon, variant would be to have a fixed filed width, and fill the number

```
float (x) as string width (w):
\ fixed field with, e.g. w=7:

1.23e-6
0.00012
0.01234
1.23456
12.3456
1234.56
12345.6
1234567
1.234e7
```

If the field width is so small that even the exponential form is not possible, it is filled with question marks.

Interpolative Formatting

There have been various implementations of elaborate formatting schemes, the *printf* from C probably the best known.

All compose the result from a string such that an escape sequence is replaced as indicated by its contents, here the percent character in C:

```
printf("Result: %3d", intval);
```

The major advantage is that the message itself is easily understood, the formatting indicators as placeholder for the values finally used. In the evolution of this techniqe, more and more features were added, and the term behind the percent sign becoming more and more complex, as the latest Python allows even exprssions inside:

```
print f"It is very {'Expensive' if price>50 else 'Cheap'}"
# instead of
msg = "It is very "
if price > 50:
    msg = msg + "Expensive"
else:
    msg = msg + "Cheap"
print msg
```

While the second form is verbose, it is easy to understand and verify, because the user needs not parse a complex string. Providing a second language for string composition while the host programming language already has one, is not considered useful.

However, string replacement provides the advantage of a quite clear template for the result string, with the clarity of formatting the placeholder values individually. This allows to translate the string later and change the order:

```
msg =: "Copy &source. to &target. failed."
msg =: string msg replace many '&source.' -> srcfn, '&target.' -> tgtfn
```

while the normal approach does not allow to have a message template catalog:

```
print "Copy", srcfn, "to", tgtfn
```

In C, interpolation formatting was essential, as composing a string from parts was laborious, and the printf functions made it easy to compose messages and convert numbers to strings in a flexible way.

The C solution uses the percent character (%) to start a placeholder which is terminated by a letter out of a given set (diouxXeEfFgGaAcsCSpnm), that also defines the type of variable. This last character (the conversion character) must precisely match the type of the expression to source the replacement, thus the replacement is entirely controlled by the format string, where the conversion character determines the way the expression is processed. Any mismatch is undefined and may result in crashing the programme.

A compatibile function `cformat (fmt) (list)` is not longer provided, as the differences were too large; a reviced formatting close to the C concept is available.

As in Python, the percent operator (%) allows to apply TAV-PL formats (not C formats) to a list of expressions. It is primarily intended for formatting single items instead of calling the corresponding function(s) directly, as string catenation is easy, even if parenthesis are required:

```
\ direct calling functions
print "Result is " _ pad left res to 12 _ " time=" _ float restime digits 6
print "Result is " _ ( '%-12;' % res ) _ " time=" _ ('%6;' % restime)
```

Revised Formatting Function

TAV-PL formats provide shortcuts for joining strings with a number of functions done by codes like in C *printf*, as are:

- minimum field length
- maximum field length (with truncation mark option)
- left or right alignment
- options for integer conversions (base, sign)
- options for float conversions (precision, sign, exponent)
- digit grouping
- padding
- quoting
- out-of-order parameters

Similar to *printf*, a field proxy starts with a percent sign, but needs to be terminated with a semicolon.

The format string is copied verbatim except field proxies that start with a percent sign (%) and always end with a semicolon. If no semicolon is found until the end of the string or the next percent sign, the characters are copied verbatim; no error is signalled. Two percent signs in a row are not a field proxy and produce a single percent sign in the output.

A field proxy has the following syntax, all parts are in square brackets to indicate that they are optional:

```
%[src:][flags][size1][.size2][flags];
```

The minimal field proxy `%;` just inserts the corresponding item from the list, converted to a string as in string catenation if necessary.

Otherwise, the elements of a field proxy are:

- source selector: if immediately after the percent sign a decimal number is followed by a colon, it overrides the source number, i.e. data is taken from the designated element of the source tuple or row, which is otherwise the ordinal number of the percent specification in the string. (The selection does not influence others that follow.) A colon without a number before is an unknown flag that is ignored.
- flags: zero or more characters that modify the behaviour of the replacement string.
- size1, size2: decimal numbers controlling the size of the string produced. If `size2` is not present, `size1` gives the minimum field width. If `size1` is not present, there is no minimal size. If both are present, the larger one is the maximum field width and the smaller one is the minimum field width, also for integer numbers; except for floating point numbers, where `size1` is the minimum field width and `size2` the precision.
- flags: again zero or more characters that modify the behaviour of the conversion. Any flag may be used anywhere.
- Semicolon that terminates the conversion (mandatory).

Flag characters may be absent or one or any combination of the following, but not a non-zero decimal digit, a dot (full stop), a percent sign or a semicolon:

```
-  pad to the right, i.e. the result is left adjusted
+  use a plus sign for positive numbers (and zero)
   a blank prepends a blank if no minus sign is present
0  pad with zeroes to the left, no padding to the right
p  pad with points (left or right)
```

```
·  pad with Unicode '·' (&middot;) (left and right)
"  quote after truncation by (string () wrap "")
q  quote for HTML after truncation by (string () quoted)
'  group the result by "'" (see 'group' function)
_  group the result by "_" (see 'group' function)
₁₀ use Unicode '₁₀' instead of 'e' in exponential format
?  mark trunction as a question mark (replace last char)
… use '&hellip;' to mark truncation (replace last char)
s  string, effectively only for documentation
d  use decimal digits for integer numbers
x,X use hexadecimal digits for integer numbers
o  use octal digits for integer numbers
b  use binar digits for integer numbers
e,E use exponential format for floating-point numbers
f,F use fixed point format for floating-point numbers
g,G use general format, i.e. fixed for small and exponential for large
i  the exponent is a SI decimal postfix (k, M, m, ..)
j  the exponent is a SI binary postfix (ki, Mi, ..)
k  the exponent is a multiple of 3 (engineering)
,  use alternate (comma) instead of a decimal point.
~  skip, i.e. ignore this tuple element (useful in localisations)
```

The result of conflicting flags is undefined. Flags not applicable are silently ignored, as are all other characters not (yet) used as flags.

The processing of a field proxy is partially inverted over *printf*:

- select the item from the list
- determine the kind of the item
- depending on the kind of the item, check if flags are present that determine the conversion to a string; if not, use the standard conversion as in string catenation
- apply general flags, if present, to finalize the string.

This means that `%s;` can be used for numbers without errror, and the `f` flag has no effect on strings.

When a floating point number is the selected item kind, `size2` is the precision, adjusted if required and used as follows:

- If the `e` flag is present, exponential is used as in `-1.2345e±98`. The precision selects the number of decimal digits after the decimal point. Because there is always exactly one non-zero digit before it, the total number of decimal digits provided is `precision + 1`. Thus, the size of the result is always `precision + 7` and padding only occurs if the minimum field width is larger. The exponent uses at least two, or three digits if necessary, in which case the total size is `precision+8`.
- If the `f` flag is present, fixed format is used, as in `-1234.56`. The precision selects the number of decimal digits after the decimal point. This format is appropriate for right-aligned numbers in a tablular arrangement where the decimal points are vertically aligned. The total number of significant digits is `exponent + precision + 1`, and the result has 2 characters more (sign and decimal point). If the total number of (significant) digits exceeds 15, insignificant digits would be required; either zeros or quite arbitrary other digits. To avoid this, the exponential format is used instead (with all 15 digits) which uses 21 characters. In this case, the the proper vertical alignment of the decimal points is (intentionally) spoiled, but avoids to display unreliable information or excessively large numbers. Small numbers that produce only a few non-zero digits (e.g. 0.01 compared to 543.21) are not shown in exponential format.
- if the `g` flag is present, a mixed or general decimal form is used, where the precision is the total number of digits to be used and the decimal point set as required. The fixed format is used (with varying number of decimal digits after the decimal point; at least one) unless insignificant trailing digits would be required (i.e. the decimal exponent greater than the precision minus one), or the number of leading zeros after the decimal point would be larger than 2 (i.e. the decimal exponent less -3), in which cases the exponential format is used. If there is no precision set, the default is six digits and no trailing zeros (as in catenation and `float (x) as string:`).

The last digit in the float formats is always rounded.

The above rules are different from the C library in some details:

- All digits produced are significant, while the C library functions may generate insignificant digits and/or excessive many digits (The number `1.0e300` combined with a `f` format produces more than 300 decimal digits that are not significant and more or less random. Switching to exponential format seems much better than producing very long strings of useless data.
- The *alternate* format, selected with the flag `#`, is always used for floating point numbers. In particular, a decimal point is always provided and at least one decimal digit befor and after it, and the result is always

evident as a (rounded) decimal fraction.

If a captial letter is used, all output letters from this element are capital letters.

Currently, only numbers use a specific conversion; all other items are converted to a string using string catenation.

Gouping uses the corresponding function that works on any string, so it applies also for pre-formatted strings with numbers.

As grouping works on digits in strings, it

The common C format %5.2f becomes %3.2f; or %f3.2;.

The mandatory semicolon terminator is the same as for HTML entity notation and is necessary even at the end of the string. Consequently, if no semicolon is found, or a percent sign found instead, the format string is returned verbatim, because no formatting took place.

The source selector is not possible in C and old Python percent formatting. Selecting a source either explictely or implied that is not in the list silently uses void instead, which is represented as the string () here (in contrast to string catenation, where it is empty).

Examples:

```
%%      provides a percent sign
%5s     use a field of at least 5 characters, right aligned
%5d     same as above, 'd' is ignored anyhow
%4:12;  select 4th element from the argument list, use 12 characters at least.
%f6.3;  use fixed point format for floats
%6.3f;  same as above
%X12;   display a number in hexadecimal format
%';     group digits with '
%_;     group digits with _
%4.8s;  Field is between 4 and 8 characters (convert to string)
%;      Convert item to string with standard string catenation
%";     Quote the string; converted to string if not a string.
```

As format () () is just a string function, when using withprint, the latter determines if a new line is made. Thus,&nl; inside a format is possible, but rather seldom.

6.4. File functions

See list of library functions.

6.5. File execution

See list of library functions.

7. Various

7.1. System variables and parameters

The runtime system creates an all-global variable $SYSTEM (similar to $COMMON) with some preset fields:

tag	kind	contents
.parms	row	parameters as provided by the command line
.env	map	environment strings
.l10nfunc	func	function used for localization of string literals
.l10nmap	map	map used if .l10nfunc is void
.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)
.randmax	integer	maximum value for simple integer pseudo-random numbers
.epsilon	float	smallest x, for which 1.0 + x ~= 1.0

.floatmin	float	smallest float number
.floatmax	float	largest float number
.floatdigits	integer	Number of decimal digits that can be rounded into a floating-point and back without change in the number of decimal digits.
.runtimeversion	string	Date and time of runtime compilation

A compiler version is not available, as different libraries may have been compiled with different compiler versions.

This variable $SYSTEM is permanently locked as the fields may not be modified by the running program.

There are two commandline options that are often used during program development:

```
--debug     switch debug on initally
--sysinfo   Runtime version and usage statistics (to stderr)
```

For convenience, the options can be set and obtained dynamically by a set of functions:

```
debug enable:
debug disable:
is debug enabled:
system info enable:
system info disable:
```

Normally, the runtime system is built such that the function

```
apply runtime options (parm)
```

is called immediately before passing control to main (parms), and providing a copy of parms with both options removed, while the original parameters are still available under $SYSTEM.parms. Thus, to nullify this feature, write:

```
main(parms):
   parms =: $SYSTEM.parms
   debug disable
   system info disable
```

7.2. Program and shell invocation

There are library functions to execute a shell command; these can be disabled if the environment variable TAVPL_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 (exec) is inhibited by TAVPL_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.

7.3. Printing

The streams stdin, stdout and stderr are automatically opened and closed by the runtime, 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.

Standard Output

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. The conversion uses the same format as the string operator _, it is quite helpful to compose an output line this way. Note the__ concatenation operator that inserts a blank.

Tuples are printed as a sequence of values, unquoted, joined by a blank. Maps and rows are printed as their number of elements in brackets or braces. All other as a question mark, followed by the hex address of the item. See ; (x) to show detailled information on any item on error output.

As a tuple is printed as a blank delimited sequence of values, a tuple can be used to print more than one value: Instead of a single parameter, a tuple can be used:

```
a =: 23
print a, 1, a + 1       \ prints a blank between the values
print a __ 1 __ a + 1   \ same result
print a _ 1 _ a + 1     \ without blanks
```

If instead of the standard delimiter (a blank) another one is desired, use the join (mixed) by (sep) function:

```
a =: 23
print join a, 1, a + 1 by '; '
```

As the join function can be used for tuples, rows or maps, all values (without keys) can be printed like this:

```
r =: 1, 2, 3, 5, 7, 11, 13, 17, 19
m =: as map string pairs 'a=9 b:"x" c=3.0'
print join r by ', '
print join m by '+'
```

To print key-value-pairs for maps, see the functions

```
map (this) as string:
map (this) as string key (delim) join (sep) quote (qr):'
```

See Formatting… for functions to compose formatted strings.

Contrary to many other programming languages, the plain print () function appends a new line character at the end, as this is the most common case. Also, string handling is easy, so it is normal to compose the output line by catenating its parts to a long string, and finally print it. As printing is anyhow time consuming for each call, the overhead of the string operations is normally irrelevant.

If a print is desired without a trailing new line, use

```
print (mixed) nonl:
```

Users might prefer the aliases printn () and printf () (), but these are not in the standard library because their names do not fit in the general pattern; might be in the library TAVPL_aliaslib.tufh.

```
printn (mixed):
   :> print mixed nonl
printf (fmt) (list):
   :> print format fmt list
```

Error and diagnostic output

If the output should be the standard error stream, use instead:

```
error print (x):
error print (x) nonl:
```

In order to insert compile time information, the HTML entity notation is expanded:

```
&lineno;    Source line number
&lno;       alias for &lineno;
&filename;  Source file name
&fn;        alias for &filename;
&function;  Template of the current function
&date;      Date of compilation in ISO notation as YYYY-MM-DD
&time;      Time of compilation as HH:MM
```

To have the linenumer prepended on an error print, use:

```
error print '@&lno;:', x        \ e.g. "@33: Hello
error print '&fn;: &function@&lno;:', x
```

As these are tedious to use, two additonal functions are a shortcut for the above lines:

```
error print (x) lno:
error print (x) here:
```

The old templates:

```
error print lno (x):        \
error print here (x):
```

are deprecated and had been removed, as the use of the local variables lno or here will result in a warning that the

local variable here is hidden by a template word.

A common use to log the kind of item supplied as function parameter might be:

```
error print x.kind here
```

The attribute .Dump (note the capital letter) supplies a (long) string with several lines containing much available information of any item:

```
x =: "hallo"
error print x.Dump
y =: ()
error print y.Dump
```

Trailing lno or here is not required, as it is already contained in the debug lines.

To print the current call stack, use the system call stack () function:

```
error print system call stack (5)   \ print 5 top stack entries
```

Debugging

As a debugging aid, the function debug print (x) (and debug print (x) nonl) suppresses the output unless the debug flag is set.:

```
debug print (x):
   ? debug is enabled
      error print (x)
```

To en- or disable debug output, a common global flag is used, that can be set and inquired by library functions:

```
debug enable
debug disable
debug is enabled
```

Unless the standard parameter evaluation is circumvented, the option --debug will set the debug flag, which is otherwise off on program start.

The debug function is called normally, including evaluation of argument(s), so inside heavily used loops its better to use:

```
? debug is enabled
   error print costly function to determine (x)
```

All error print functions are also available with error replaced by debug:

```
debug print (x) lno:
debug print (x) here:
```

Activating the debugging output does not require recompilation; either the --debug command line option (if not circumvented) or the function debug enable will do so.

If the simple conditional compilation (via the -e compiler option) is used, debugging can be enabled at compile time:

```
\~  debug print a, b, compute a with b
\~  error print a, b, compute a with b
```

7.4. 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) and (y):
   ? y = 0            \ mathematical definition of gcd
      :> x
   :> gcd (y, x %% y)
```

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) and (y):
   ?*
      ? y = 0            \ mathematical definition of gcd
         :> x
      \ :> gcd (y, x %% y)
      x, y =: y, x %% y
```

Tail recursion, i.e. calling a function inside its body, is not too difficult to detect, and is presumed to be fairly easy to implement, thus the programmer might use it freely.

8. 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 TAV-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 \$. It is copied to header files and honored by \^.

In particular, the line is copied to the C source code, with \$ replaced by //$$. The script to compile and link the main function serches for lines beginning with //$$, and supplies the rest as additional linker options. For TAV 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 TAVPL crypt.c may not only need its object file, but also -lcrypt. Thus, the source code shall contain

```
//$$ -lcrypt
//$$ TAVPL_crypt.o
```

Of course, the header extraction program TAVP_genhdr has to copy both forms to the header file.

External C functions

As TAVPL compiles the source into a C (or C++) function, linking with functions written in C is possible, provided

that these functions behave like a TAV-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 TAV-PL code.

Modules

TAV-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, TAV-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 TAV-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 TAV-PL, a header file should be created because - packaging is required anyhow - the header reflects the library

Recommended extensions are .tuf for TAV-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 map:

```
set string translation map (ary)
```

The creation of the map may be done by map literals or reading configuration files. The parameter must either be a map 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 map.

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 map 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 map, an interface to *GNU gettext()* may be provided, possibly by allowing to register a callback for translation.

Binding to existing libraries

To use existing libraries with C functions, each such function has to be translated in a wrapper module, that represents the respective library — as usual.

As this requires items to pass data structures of the foreign library, system items are provided, that cannot be modified in a TAV-PL programme, except via special libraries.

This has successfully been tested so far only for two graphic libraries (Cairo and SDL), as these do not require callbacks.

While TAV-PL does support callbacks via function references, two problems are difficult to master:

- there must be wrapper functions for the TAV functions called
- often, callbacks are not properly discarded and could result in memory leaks
- the user supplied parameter is often just a pointer, and proper memory management is extremely hard, if not impossible

9. 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 only allow good programs from an object oriented design.

TAV-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 TAV-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 TAV-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 keyed aggregation
- fields, called attributes, with any contents

the first question is assumed to be true.

Note that constraints are not applied to bunches that are locked.

How to do object oriented programming

Coding based on an object oriented design in TAV-PL might use — as shown in the libraries provided — the following design pattern:

- An object is named with a word or a sequence of words
- Functions associated with the object use the word(s) first, then the object reference as parameter, then the rest of the template
- Objects are created using factory functions

Although the word new is not an operator, factory functions usually start with the wordnew followed by the object name, followed by a trailing template. The documentation processing ignores leading word new in creating the sorted key. For example:

```
new hashmap with (n) slots:
   this =: [n]
   this.tag =: 'hashmap'
   ...
   :> this
hashmap (this) get (key):
   ....
```

No conflicts arise, e.g. with

```
file stream (fd) put line (line)
command stream (fd) put line (line)
```

The template matching is quick and straight forward, and the function templates are verbose anyhow.

In traditional programming languages that use single words for function names, an underscore (or CamelCase) is often used, e.g.

```
file_stream_put_line(fd, line)
command_stream_put_line(fd, line)
```

In particular together with the traditional functional notation, this is hard to read, even if underscores and camel case is combined:

```
fileStream_putLine(fd, line)
commandStream_putLine(fd, line)
```

and it is much more readable to write:

```
fd = fileStream("my file");
fd.putLine(str):
```

which also allows to change the object, provided this does not create subtle errors.

In TAV-PL, the word new is not an operator, and factory functions must be used. To call a function logically associated with a bunch, a double colon is used:

```
fd =: new file stream ...
fd::put line str
```

If a compiler supports the notation, a more complicated template matching is used, that starts after the first parameter (in the template) and matches the rest of the template. If the match is unique, the function is found; otherwise, for each of the leading words, a test is made, like in:

```
? fd @= 'file stream'
   file stream fd put line str
|
   ! fd @= 'command stream'
   command stream fd put line str
```

Due to the verbose way that the tail of the template may be written, the number of matches can be held low, if so desired. Morover, the dynamic function selection is in most cases not relevant for the performance.

Note that while there is no functional need to forbid both notations, i.e.

```
fd =: file stream reads 'my.file'
file stream fd put line str
fd#put line str
```

If ???constraints are implemented, the runtime tag tests are seldomly needed, at the expense of more complex progamme writing.

In order to restrict this extension, the function can be defined using the double colon instead of the (this) parameter, and the scoll symbol in the body instead of the object reference:

```
hashed map:: put (str) at (key):
   i =: string key hashed %% @.count
   m =: @[i]
   m{key} =: str
```

and to avoid repeating the prefix:

```
hashed map::
   put (str) at (key):
      i =: (string key hashed) %% @.count
      m =: @[i]
      m{key} =: str
   get (key):
      i =: string key hashed %% @.count
      m =: @[i]
      :> m{key}
```

OO outdated section

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 TAV-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 TAV, 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 allow classes by naming bunches
- constraints (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
```

Although the tags are strings, 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, and only for rows and maps, because they are mutable. Tag names that are kind names, i.e. integer, rational, float, string, boolean, function, are reserved and cannot be set, see .Class atribute.

Thus, string functions can be defined as follows:

```
string:: clip upto (n):
\ instead of
string (str) clip upto (n):
use this:
   s =: 'hallo, hallo'
   print s.clip upto 5
```

Instead of the two lines:

```
r =: new row
r.Tag =: 'myRow'
\ or
m =: new map
m.Tag =: 'myMap'
```

the factory function acceps a string paramter for the tag:

```
r =: new row 'my Row'
m =: new map 'my Map'
```

which is only useful as setting the Tag is better visible. The bigraphs{} and '[]' cannot be used for this, as any item inside is assumed to be a tuple to set the bunch initially.

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'     \ factory function sets Tag
! 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 TAV 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 TAV 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 TAV-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 TAV 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)

10. Examples

More examples can be found online at http://rclab.de/TAV-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:

```
linked list give (first):
    ? $ =~ first        \ if initial call
    $ =: $.next         \ 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
```

11. 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.

Memory shortage interception

Some functions might return a fault item if there is not enough memory available.

In particular, recursive function activation might result in a fatal runtime error terminating the program immediately. While theoretically a function in this case could return a fault item, it seems to be hard to catch this case on C level already.

Most probably memory is exhausted by catenating two (very) long strings; which currently is a fatal runtime error.

In any case, avoidance and interception of lack of free memory is still to be analysed and integrated thoroughly.

Anonymous functions

Anonymous functions use the backtick notation, but do not start with a word like any other function template, but with an optional chain of parameters in parenthesis, followed by a colon and a statement:

```
new hashmap size n hash `(x): :>x`      \ use key as hash
system on exit call `: $primes =: ()`   \ clear global
inverted =: `(a) (b): :> compare (b) with (a)`  \ reverse parameters
```

So it looks like a funktion template without words. This notation seems clumsy, but using a list or sequence of parameter words would be inconsistent.

Conditional expressions are useful here, e.g.

```
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 nor time; it is just a syntactic extension which is most useful in combination with the condensed notation or when a simple callback function is required.

The function is created were the statement lexically occurs in the prgram text, thus created only once in a loop.

```
r =: []
?# i =: from 1 upto 3
   r[] =: `(x): :> x + i`
! r[1] = r[2]              \ same function references
r1 =: r[1]
r2 =: r[2]
! r1:(5) = 5       \ not 6
! r2:(7) = 7       \ nor 9
```

The part after the colon is treated like any function body, and the compiler should complain about using without prior assignment; as void is zero in additions, just the argument is returnded.

Syntactically it would be possible to have multiline bodies, and then allow more than one statement and the condensed notation:

```
max =: `(a)(b): ? a > b
              :> a
              :> b`
max =: `(a)(b): ? a>b; :>a; | :> b`
```

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 map with uniform references can be internally optimised, in that a map of bytes is kept contiguous internally and just flagged in the map header correspondingly.

Condensed notation

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.)

If during statement scan, a semicolon (;) is encountered the statement is considered complete, and the rest of the line treated a another line with the same indentation level, after removing any white spece after the semicolon:

```
x =: 0; y =: 0
tok =: toks[ti]; ti =+ 1
```

As the semicolon has no other function, this kind of line splitting can be done context independent.

If the statement terminated by a semicolon requires a dependent block — determined by the leading symbol ?, ?* or ?# — the rest of the line is treated as indented one extra level as by a single space:

```
? a = (); a =: []                \ better use a =~ []
? i < 0; ?^                      \ conditional repeat loop
? i > 0; ?>                      \ conditional terminate loop
?* x.next ~= (); x =: x.next       \ find last in chain
? is x fault; x.checked =+ 1; :> x  \ propagate fault up
```

They can be nested and replace the shortcurt &&:

```
? x ~= (); ? x = 5; print x
```

Alternatives may use the usual signs ?~ or |:

```
? x > y; z =: x;| z =: y
```

Note that ;| is not a bigraph; the semicolon splits the line unconditionally and is never part of a bigraph; after a line split, the line is checked by inspecting the first token if it requires an unindent one level. These tokens are |, ?~, |? and ~?.

The condensed notation is useful in short anonymous functions, once line breaks are supported inside anonymous functions:

```
max =: `(a) (b): ? a > b; :>a; | :>b`
```

Because the dependent statements are not limited to assignments or function calls, two conditional statements may be nested, e.g.:

```
? x ~= (); ? x.attr = val; :> x
\ same as
? x ~= ()
   ? x.attr = val
      :> x
\ or
? x ~= () && x.attr = val; :> x
```

Conditional expressions

Conditional expressions use ?! (if true) and ?: (if false), enclosed in special parenthesis ?(and ?):

```
x =: 3 + ?( condition ?! trueval ?: falseval ?)
```

The condition is a boolean expression, and trueval and falseval represent expressions that are allowed on the right hand side of an assignment.

The start symbol ?(looks superfluous, but clearly indicates that a boolean experession must follow, and the whole conditional expression is normally in parenthesis anyhow, at least for clarity:

```
x =: math sqrt ?( y < 1.0 ?! 2.0*y ?: y^2 ?) + y
```

The shortcut symbols && and || in logical expressions are essentially conditional expressions:

```
lhs && rhs == ?( lhs ?! rhs ?: ?-  ?)
```

```
lhs || rhs == ?( lhs ?! ?+  ?: rhs ?)
```

It looks a bit clumsy, but clearly shows the borders and is thus easy to be scanned by the programmer.

Even in C mode, the equivalent is not (yet?) supported due to implementation problems:

```
\ x =: math sqrt (y < 1 ? 2*y : y^2 ) + y
```

Slices

It is not intended to include slices, in particular string slices, as a language concept as it is neither necessary nor useful.

The typcial Python string slice is:

```
str[start:end:step]
str.slice(start, end, step)
```

is provided in TAV-PL via functions:

```
string (str) from (start) upto (end) step (step)
string (str) from (start)
string (str) from (start) length (len)
```

or the class functions

```
str::from (start)
str::from start length
```

Allowing tuples or triples for

could some day be implemented as tuple index to a string, i.e.

```
str[start, end]
str[start, end, step]
```

replacing the respective string function

Nevertheless, some considerations follow.

When using the double dot bigraph, it cannot be used for line contiuation.

As opposed to tuples — which are an indexed collection of individual items — a slice is defined by a range of values, defined by the first and last and a rule to compute the values in between. Consequently, ranges of integer numbers are normally used.

The slice bigraph is a double dot, e.g. a range of integers:

```
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 are ordered lists of characters, the slice might be useful here:

```
s =: 'The quick brown fox'
t =: s[5..9]        \ instead of: string s from 5 upto 9
\ or s::from 5 upto 9
! t = 'quick'
```

It would be useful only if true string slices are implemented, allowing a string buffer (with a use count) to be shared by string references. While this is possible because strings are immutable and the current implementation uses detached string buffers, but not for all strings, not for strings with a few characters only and string literals.

Row and map literals

Because of their dynamic nature, rows and maps are normally created at runtime and filled from data calculated or read from files. Thus, there are conceptually no row or map literals.

As tuples are like (small) rows with fixed contents, they can often be used when a row literal were desired:

```
primes =: 2, 3, 5, 7, 11, 13, 17, 197
```

If a row is required, because fields are used or it must be changed, use the library function or enclose the tuple in brackets:

```
rowOfPrimes =: as row 2, 3, 5, 7, 11, 13, 17, 197
rOP =: [2, 3, 5, 7, 11, 13, 17, 197]
! rowOfPrimes = rOP
```

As a map from a tuple requires a pairs of key and value, there is a function `tuple (tup) as map by pairs`, available as alias, or by enclosing the :

```
amap =: as map ('a', 9), ('b',"x"), ('c', 3.0)
amap =: { ('a', 9), ('b',"x"), ('c', 3.0) }
```

This string can be generated by

```
map (this) as string key (delim) value (sep) quote (qr):
map as string:
   :> map amap as string key ',' value ('(', ')') quote ()
```

As the syntax is hard to read, pairs (tuples with two elements) may also be created by the binary operator `:>`:

```
amap =: { 'a'-> 9, 'b' -> "x", 'c' -> 3.0 }
```

Values for duplicate keys replace the earlier ones; due to the small number, it is not worth adding without check for multiple keys.

As there are no genuine row and map literals, named literals cannot be used to create rows or maps at compile time; but tuples are:

```
#aMap =  'a' -> 9, 'b' -> "x", 'c' -> 3.0
#rowOfPrimes = [2, 3, 5, 7, 11, 13, 17, 197]
#mapByPairs =: { a' -> 9, 'b' -> "x", 'c' -> 3.0}
```

Field change callbacks

In object-oriented languages, object fields 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.

In TAV-PL, a field 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[14].

Instead of

```
obj.set_width(5)
x = obj.get_width()
```

this is just a normal assignment in TAV-PL:

```
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 =: new map '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 field; for a getter function, the value returned is used instead of the hidden value of the field. Note that during the call of any of these two functions, normal access to fields is used.

Not that comparison is done with >= and <=, so there is no collision[15].

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 field 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

To allow callbacks not only for fields, but also for the element access, i.e. row[key] and map{key}, the syntax

```
obj.> =: `get (this) from (key)`
obj.< =: `put (this) at (key) value (val)`
```

will finally be the right way to define it.

Currently, two controls are used, .OverrideGet and .OverridePut, that can each hold a reference to a function with two resp. three parameters. The first one is called whenever the row or map is accessed, i.e. the form row[key] or map{key} is found within an expression, and should return a value, while the second is called if the form is the target of an assignment. Both have the bunch reference as first parameter, the key as second one, and put the value to be set as third one:

```
matrix (this) get at (keys):+
  i =: matrix this calculate keys
  :> this[i]
```

When such an override function is called, both override functions are disabled for the bunch until it returns and thus can access the row or map normally.

The overriding access functions should not be called otherwise to prevent circular invocation, i.e. not exported.

Both attributes are stored like a field, but with the names :OverrideGet and :OverridePut which are protected from accessed as fields due to the leading colon.

Overriding of operators

Like overriding of bunch access, such a mechanism allows to override other operations, in particular operations in expressions like add, subtract, multiply and divide, for maps and rows.

Implementation is fairly easy, if factory functions are used. A factory function for a (tagged) bunch sets a???fields with a function reference, and the operator code checks if the (left hand side) operand has such a control field, and then calls the function.

The syntax to establish complex numbers is then:

```
new complex number (re) i (im):
  this =: {}
  this.tag =: 'complex number'
  this.re =: re
  this.im =: im
  this.OperatorAdd =: `complex () add ()`
  this.OperatorSub =: `complex () subtract ()`
  this.OperatorMul =: `complex () multiply ()`
```

```
  this.OperatorDiv =: `complex () divide ()`
  this.OperatorNeg =: `complex () negate`
  this.OperatorAbs =: `complex () abs`
  this.OperatorEqual =: `complex () equal ()`
  \ no comparison
  :> this
complex (x) add (y):
  \ no shortcut to return y if x = 0.0, because equals comparison
  \ is not supported, although this might be a useful case
  :> new complex number x.re+y.re i x.im+y.im
```

As usual, the operators combine the two arguments and return a complex number.

The problem with this solution is that attributes could be function references, but are not automatically called; unless the specification for fields and attributes defines that number attributes are not deliverd, but called as unary functions.

It seems to be fairly inefficient to call function references instead of calling directly the corresponding functions; however, this overhead is the overhead of finding the control field values. Together with the storage requirements to save the latter, the overhead is nevertheless quite small, but tests are still pending.

Overriding of string conversion

As there is a default conversion to a string for numbers (and booleans), a special attribute.ToString is justified, that, when set with a function reference, calls a function with one argument that converts the item to a string in default format.

However, for common items this is already done by string catenation; for items of larger complexity, such a string might quickly become

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 pair:

```
?# x =: find re 'a[0-9]+b' in s
 ? x.1 > 0 & x.2 > 0
  x =: clip string s from x.1 length x.2
```

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, there is no loss in speed or size. 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.

Combining (join) bunch elements to strings

In particular in debugging, it is often required to assemble all values (or keys) of a bunch to a string.

Serveral library functions allow this:

```
row (row) join values
row (row) join values by (sep)
row (row) join keys
row (row) join keys by (sep)
map (map) join values
map (map) join values by (sep)
map (map) join keys
map (map) join keys by (sep)
```

No separator parameter delimits by a single blank; for consistency (and for future optional parts), a void separator does the same; to catenate without separator, use the empty string.

Normally, shortcut version are used:

```
join (bunch) values
join (bunch) values by (sep)
join (bunch) keys
join (bunch) keys by (sep)
join (bunch)                \ same as join (bunc) values
```

To print all values of a row or map, write

```
print join rmt
```

To print all values of a row in separate lines, use

```
print join row values by '&nl;'
```

Several scan functions are provided in this context:

```
row (row) give keys         \ ascending
row (row) give values       \ ascending keys
row (row) give values ascending keys
row (row) give values descending keys
row (row) give values ascending keys using (func)
row (row) give values descending keys using (func)
map (map) give keys             \ undefined order, with multiples
map (map) give keys ascending       \ without multiples
map (map) give keys descending
map (map) give keys ascending using (func)
map (map) give keys descending using (func)
map (map) give values               \ undefined order of keys
map (map) give values ascending keys
map (map) give values descending keys
map (map) give values ascending keys using (func)
map (map) give values descending keys using (func)
```

Lazy versions that determine the kind dynamically:

```
give (bunch) keys               \ order depends on bunch
give (bunch) keys ascending
give (bunch) keys descending
give (bunch) values             \ key order depends on bunch
give (bunch) values ascending keys
give (bunch) valuse descending keys
keys of (bunch)                 \ short for 'give (bunch) keys'
values of (bunch)               \ short for 'give (bunch) values'
```

Some variants may not yet be present; and not all are defined here in order to avoid cluttering the list with rarly used variants, e.g. giving row keys in descending order, that can be done normally by:

```
?# i =: from row.last downto row.first
   ? row[i] = ()
      ?^
   ....
```

or as a scan function:

```
row (row) give keys descending:
   ? $ = ()
      $ = row.last
   |
      $ =- 1
   ?*
      ? $ < row.first
         $ =: ()
         :> ()
```

```
    rv =: row[$]
    ? rv = ()
      $ =- 1
      ?^
:> rv
```

Output to standard output (print)

Initially, there are only two print statements:

```
print item
print item nonl
print partial item   \ deprecated
```

where item could be a string or a number, and a complex line is created by using string catenation (and __).

Instead of the negation nonl, the forms

```
print item nl
print item
print nl
```

would be more systematic; but printing an entire line is more often needed, as in:

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

A simple extension was to support lists with a standard delimiter:

```
print (list):
   ? is list list
      print as string join list by ', '
      print join list by ','
      print join map xx values by ',

print list
```

If a different delimiter is required, use the join function.

Multi-dimensional maps and rows

As bunch values may be bunches, multi-dimensional rows or maps can be created by assigning rows or maps to the elements (known as Iliffe Vectors).

Basically, the secondary bunches are created in advance, as in

```
a =: []
?# i =: from 1 to 11
   b =: []
   ?# j =: from 1 to 12
      b[j] =: i*j
   a[i] =: b
! a[3][4] = 12                \ if the compiler allows stacked keys
```

An example would be a row of maps, the example filling it from a file keyed in a row by length, and counting each word:

```
r =: []
?# w =: give lines from standard input
   x =: r[w.count]
   ? x = ()
      x =: {}
      r[w.count] =: x
   x{w} =+ 1
```

As map keys can be any items, they can be tuples in particular:

```
map =: {}
map{1, 'a'} =: 1, 'a'
le =: map{1,'a'}
! le = 1, 'a'             \ if the compiler supports tuples here
```

Multi-dimensional rows are thus rows which are keyed by tuples:

```
row =: [5,5]              \ allocates 25 cells
?# i =: from 1 upto 5
```

```
? j =: from 1 upto 5
   row[i,j] = i+j
```

or, if tuples are supported by a generator scan:

```
?# i, j =: from 1,1 upto 5,5
   row[i,j] =: i + j
print row              \ if printing of bunches is supported
```

The attributes .first and .last still refer to the underlying row.

If the row is multidimensional, the attribute .dimensions has a reference to the list of maximum elements per dimension. If .origin is set, it may be a single integer or a list of integers, one for each dimension. If void, integer 1 is the default. The origin of each dimension is subtracted from each dimension in calculating the offset, then the linear cell computed with zero origin.

While it may be changed at any time, this will seldomly be useful and thus may only be supported before the first cell accessed.

Changing the number of elements per dimension normally results in heavy data movements, and may thus be rejected (or because the effort to write the necessary code was not yet justified).

Tuple subcodes

If libraries prefer to use tuples instead of bunches, a limited number of tags (<250) can be used. The tags should be qualifiers like tufpl.complex or de.glaschick.libxx.xx.

The system keeps the qualifiers in a map and internally uses a handle.

A tuple with a tag can be created from an existing one by:

```
nt =: system tuple (tup) set tag (qualstr):
! system tuple nt get tag
```

Note that tuples are immutable, so there is no means to set the tag.

Extended operators

For the use with libraries, the common operators+, - etc may be registered for user extension. Befor an operator results in a fatal error because the operands are unknown, a set of functions is called one by one with the operands. If void is returned, the next one is tried; otherwise the returned reference is the result of the operator. The order of calling is undefined; in particular, the last successful one may be called first (least recently used).

Adding functions is additive, i.e. several functions may be registered. The last paraemter (tag) is a string identifying the function; if it is void, the function reference is removed.

Operator functions must be registered by:

```
system operator binary (op) add (funcref) tag (tag):
system operator unary (op) add (funcref) tag (tag):
```

The parameter (op) is one of +, -, / etc.

The function is called with the operand(s) and must determine if it can process these, and return void if not. Such an operator cannot supply void as a result.

This means that a library must be actived by an initalisation call.

Template permutations

Instead of parameters by keyword, TAV-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.

To use any parameter position to start a permutable part is conceivable, but may require stronger selection by the words until the first parameter (used for class functions anyhow).

There is no urgent implemetation need, as the same effect can be obtained by writing transfer functions:

```
from (a) step (b) upto (c):
   :> from a upto c step b
from (a) upto (c):
   :> from a step () upto c
```

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.

Memory administration as used in TAV-PL is not thread safe. If items are shared by threads, incrementing and decrementing the use counter must be atomic. Also, the queue of unused items to speed up allocation must be thread-local, unless the overhead of mutexes is payed.

As both thread execute instructions (not statements) in an unpredictable order (unless synchronisation primitives are provided and properly used), the execution sequence of such a programme becomes rather 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 scan 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 a scan loop, and defining that extra parameter if a function contains the scan context symbol. It would create a special class of scan functions.

Using only bunches as globals and the `.lock` and `.Revisions` attributes, a high degree of thread-safety can be achieved, provided that .lock and .Revisions are changed atomically.

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 functions

The current scheme for scan functions is simple, yet proved so far to be powerful and useable, however:

- The function prototype does not syntactically indicate that the function is a scan function, thus it is not checked if the scan loop assignment calls a scan function.
- If threads are to be provided — which is debatable, seeThreads, but might be unavoidable — the scan context must be a thread-local variable or an extra parameter.
- Often, the structure of a generator function is easier to understand

Checking that a function called in a scan assignment is finally a scan function is not primitive and not justified, because other pitfalls were more common. Also, this error is detected very early when testing.

Thread-local storage is available in C, although the performance penalty is unclear.

Alternatively, all scan functions could have an extra parameter automatically provided at the point of call and carrying the scan context. Either indirect scan functions would be no longer allowed, or this parameter would be used by any function call. Note that TAV-PL until now does not have a indirect item; in all cases so far either a bunch or a tuple (of one element) are used.

The latter may be necessary when introducing threads anyhow.

Generator functions are often shorter and clearer if yield is used, e.g. using the #> digraph:

```
give Fibonacci numbers upto (n):
   a =: 0
   b =: 1
   ?* a <= limit
      t =: a
      a =: b
      b =: t + b
      #> a
```

or in tuple split notation:

```
give Fibonacci numbers upto (n):
   a, b =: 0, 1
   ?* a <= limit
      a, b =: b, a + b
      #> a
```

compared to the scan function:

```
give Fibonacci numbers upto (n):
   $ =~ 0, 1                  \ initial tuple if first call
   r =: $.1 + $.2             \ next number
   $ =: $.2, r                \ save previous and current value
   ? r > n
      $ =: ()                 \ end of scan
   :> r                       \ return next number in any case
```

Another example:

```
\ generator
give primes upto (n) :
   pt =: [n]               \ table preset with void
   #> 2                    \ first prime
   p =: 3                  \ next prime
   ?*
      \ search in table
      ?* pt[p] ~= ()
         p =+ 2
      ? p > n
         ?>                \ done
      #> p                 \ yield next prime found
      \ mark multiples
      ?# j =: from p*p upto n step 2*p
```

```
        pt[j] =: p          \ mark all multiples as non-prime

\ scan function
give primes upto (n) :
   ? $ = ()              \ first call
      $ =: [n]           \ table preset with void
      $.idx =: 3         \ start here at next call
      :> 2               \ suppply first prime
   p =: $.idx            \ continue here
   ?* $[p] ~= ()         \ skip non-primes, always finds void
      p =+ 2             \ $ not used in scan body
   ? p > n               \ done?
      $ =: ()            \ yes, clear scan context
      :> ()
   \ prime found, save continue position
   $.idx =: p + 2
   \ mark multiples
   ?# j =: from p*p upto n step 2*p
      $[j] =: p          \ mark all multiples as non-prime
   :> p
```

However, generators in general are as difficult to implement as coroutines, because the stack frame of the generator has to be saved by the yield, which requires some fiddling in the internal structures in C, normally done be assembly instructions that are not really portable.

Using Duff's device (it is allowed even to jump into loops in C), yields could be implemented fairly portable, but the problem of saving local variables is still unsolved.

Switches

There is no equivalent to the C switch control. It was relevent in early times as an efficient multi-way branch, which is no longer necessary. Also, it is rather restricted in application and rather tricky, thus confusingly, by its fall-through cases.

Instead, a chain of conditions has to be used:

```
? x = 1
   do something
|? x = 2
   do otherthing
|? x = 3
   ...
|
   done otherwise
```

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, not restricted to equality comparisons and can be ordered for performance.

Alternatively, a map with references to functions can be used.

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 TAV-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 an error

- A placeholder in the pattern not used in the replacement is not 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:

```
\\ <\x> \\ '\x="' _ \x _ '"'
\\ <\m{\i}> \\ '\m{' _ \i _ '}="' _ \m{\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 \\ |?
\\ while \\ ?*
\\ 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 and in particular Simon Tatham's explanation) are specified in a generalised message passing model, that allows bidirectional value (message) passing.

As scans are light uni-directional coroutines, it is unlikely that full coroutines will be implemented.

Coroutine send and receive is indicated by the digraph <> at either the left hand side of an expression (send, yield) or as a term in an expression (receive)

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:

```
func1:
```

```
   ?# x =: <~          \ receive
      ~> x*x           \ send
func2 (p):
   i =: 1
   ?* i < p
      ~> i
      j =: <~
      print i _ '^2 = ' _ j
 func1 <> 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 most implementations, as a stack suspend operation is requiered. 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.

Note that scan functions are similar to coroutines, see below in??? functions as coroutines.

Unified Rows and maps

The distinction between rows and maps was initally very strict, and it deemed useful to distinguish both syntactically, in particular as using integer numbers for larger maps was inefficient, which is technically solved now. Logically, rows are optimized maps and could even be defined by constraints. The are addressed by a decent range of integer and never shrink by itself, but grow on demand as maps do.

While (basic) maps shrink dynamically when cells are no longer used, it is unclear how this is done for hashed maps.

So the distinction between (square) brackets and (curly) braces is n, in particular as the creation of maps and rows use factory functions now. This frees braces for other usages in future, e.g. as parameters in function templates.

If rows with their different, more efficient access mechanism were completely removed, a significant number of applications would suffer from relevant increase of processing time, even if the maps use hash tables.

If the access mechanism would be automatically switched if the key is an integer, applications having sparse integer keys would have an increase in memory usage that might even force them to convert integers to strings as keys to use maps.

This finally means that creating a map by `new map` is the universally applicable kind, and `new row` a map restricted to (small) integer keys. A new kind `box` as unified item is not useful.

On the contrary, the classes `row`, `fault` and `chunk` are considered special variants; i.e. the kind `map` will have the subkinds `row` and `fault`, keeping the class names.

The migration steps from 0.9 via 0.10 to 0.11 are:

- use brackets for map access, and later mark braces deprected
- check the portfolio of code for changes.
- implement change to and expand of hashed keys for general maps
- check the list of functions for rows and maps for overlaps and differences.
- change the code for a row as a subkind of a map and remove redundant code
- same for fault as a subkind of a map

As a first step, square brackets may be used for maps, and curly braces are allowed in function templates instead of round ones.

Following are two proposals for:

- a row storage that avoids copying when expanded, and leave huge gaps unallocated
- a map storage switching dynamically to hash tables

Revised row implementation

To use the new row storage system for a fairly large hash table for the maps is not useful, as any decent hash function scatters the keys uniformly such that there are no large gaps, and all storage is allocated anyhow.

The revised row storage no longer uses a single block of memory for all keys, which is expanded by re-allocating the space between the smallest and largest index.

It allocates storage in fixed morsels, e.g. 64 references. Whenever more are needed, an index table is used that has links to the basic morsels as needed. This slows down the access time slightly, as the index is split into the lower 6 bits for the morsels and the next 6 bits for the index. If the indices are gradually expanded at the end, every 64 expanding keys a new index table is added. When all such 4096 elements are used, another top level index block is used, upto 2^{16} = 262'144 elements, and having a slightly larger memory overhead. Next step are 4 levels for 16'777'216 elements.

Of course, the granule can be smaller, e.g. 16 or 32, or even larger, 128 or 256. The larger granules will increase memory requirement for many small rows, while the smaller ones do increase access time. Taking into account that a row item has uses 11 words on top of the key tables, the proposed granule of 64 elements seems sensible. Implementaion will have the granule as compile time parameter anyhow, thus benchmarks can be done. Changing the granule will have no perceptible effect in nearly all cases.

This scheme avoids copying larger blocks of memory and does not allocate memory for large gaps, but its merits are small compared to the already implemented version. Using it for the hash tables for maps is not sensible, as the hash function should distribute faily equal over the table and thus enforce full allocation anyhow.

Revised Map Implementation

The revised maps uses linked lists, i.e. chains of elements, for fields like in rows. These require 2 words in addition to the 6 words used for the item itself.

While the memory use is quite acceptable, the quadratic increase in time when inserting a new element is not.

Thus, at least after a certain number of elements, either binary trees or hash tables should be used. The former require order comparison and are thus not used.

The latter depend on a fairly well distributed hash function, that gives equal hashes for equal items and is provided by the `hash of ()` function:

- for strings, the function `string () hash` is used
- for integers, the 32-bit remainder of the value is used.
- for quotients, the sum of both parts is used like an integer
- floating point numbers are still not supported
- for all other items, the (folded) memory address is used, as these are not equal unless the same.

Floating point numbers are not allowed, as they cannot be compared for equality.

As the number of elements vary from a handfull to several millions, a dynamic expansion is required, that does not slow down the insertion of a new element significantly if the table is expanded.

This excludes hash tables with serial overflow (searching the next free slot linearily), as with this — storage efficient — solution the whole table must be reorganized at once.

The revised map uses linked lists not only for collision, but also to keep the hash tables small and to spread the reorganisation time to several table accesses. The memory overhead is not small, but the scheme is robust, independent of the quality of the hash function. Also, when the hash table is replaced by a larger one, the task of re-hashing can be distributed over several calls.

The costly operation in seaching a linked list is the key comparison, which includes checking the kinds of the items. Thus the search time is approximately reduced by the number of slots as a factor, while the hash table has the overhead to calculate the hash value of the key. As every element is stored in a — perhaps short — chained list, there is always an overhead of two words in addition to the one holding the (reference to) the value item. A hash table adds as many words as there are slots; when it is expanded with two elements per slot (on average), the overhead is initially 3.5 words instead of 3 words per entry. If next expansion takes place with about 4 entries per slot, the overhead is only 3.25 word.

Up to 64 elements, a single chain is used which is reasonably quick and has low overhead. If this limit is reached, an initial hash table with 64 slots is created and all existing entries are re-hashed to the new table, so that on average there is one entry per slot initially.

This intial hash table is used until the average chain length is 16, i.e. when about 1024 elements are in use.

Then, a new table four times the size, i.e. 256 slots, is allocated. The old hash table is kept for a while. In any case, an entry is first hashed and located in the actual table. If found, it can be either read or replaced or added

without further inspection of the old table, as the keys on both are never the same. If not found, it may be still in the old table. When it is a read operation, the value is the desired one, and the entry moved from the old to the actual table. Otherwise, the new value is already in the actual table, so it must be removed from the old table. Note that there only one hash calculation, because the slot number is simply the hash value modulo the slot size, for both, the actual and the old table. By this strategy, the time to re-hash entries is distributed over the following key searches; and writing a new value does never require significantly more time because the hash table is reorganised.

This table is now sufficent for upto 4k entries, when the expansion by the factor 4 is applied again:

```
Level   Slots   max Entries
0       --      64
1       64      1ki
2       256     4ki
3       1024    16ki
4       4096    128k
5       16384   512k
6       65536   2Mi
7       256ki   8Mi
8       2Mi     32Mi
```

Note that every entry requies 40 Bytes for the item and 16 Bytes for the entry in the linked list, thus in level 8, the table alone used 1.8GiB, plus the data, if it are not numbers or short strings.

When enlargeing, the old table has to be cleared, before it can be replaced by the current one as old one, and a new table created. Either its elements could be inserted into the current one, or already into the new one. The latter needs slighly more space, as for a short time, all three, the old, the actual and the new tables are needed; but it saves time.

The example unixdict.txt has 25k words with 206k bytes, requiring more than 175kB for the storage. On level 4, we need 1024+4096=5120 words or 30kB for the old and the new table, which is considered acceptable.

This file is demanding as it is already sorted (testing the distribution of the keys) and has unique words. When all these are read in into a map, the old hash tables are still rather full, so that on enlargement, they have to be re-hashed (unless the space for another word is spent.)

For the hash functions, it is indispensible that items that are compared equal deliver the same hash value (while collisions, i.e. the same hash for unequal items, are unavoidable). The former means that for immutable items, i.e. integers, strings and tuples, the contents must be hashed. As bunches (maps and rows) are only equal if the item reference is the same, the latter can be used as hash.

For integers, it is the absolute value (modulo machine integer size for big integers).

For strings, the hash is an unsigned integer number calculated iteratively by

```
sum =: (sum * 127 + byte[i])
```

where byte??? i is the i-th byte (not code point) and the sum is taken modulo 2^{17} or 2^{18}. Its hash calculation is quick, but still linear with the number of characters; however, it is not exprected that extremely long strings are used as keys.

Real numbers are still barred as keys.

If tuples are equal, all the elements are equal, thus the (weighted) sum of the elements hash (modulo some large integer) seems to be a sensible choice, even it this means that tuples as keys (not as values) for maps use slightly more cpu time. As other items are equal only if identical, i.e. having the same memory address, it can be used as a hash already, maybe folded or shifted or multiplied by a prime number like 991.

The revised map should now maintain the attributes.First and .Last for all integer keys (void if no integer key), allowing to scan the integer keys efficiently.

It might be considered to have controls to set the hash function and the compare function, opening a door for real numbers as keys.

12. Features diapproved

Some proposals or former specifications were abandoned and are kept her in order to show that the feature was thoroughly checked before not accepting it.

C compatible Percent formatting

The syntax is compatible to the traditional C printf family; i.e. it differs in some points that may be signifcant.

The pattern string is copied verbatim except format conversion specification, that start with a percent sign (%) and end with a conversion specification character. A conversion specification has the following syntax (parts in square brackets are optional]:

```
%[src:][flags][width][.precision]convspec
```

Two percent signs in a row are not a conversion specification and produce a single percent in the output.

The format is:

- source selector: if immediately after the percent sign a decimal number is followed by a colon, it overrides the source number, i.e. data is taken from the designated element of the source tuple or row, which is otherwise the ordinal number of the percent specification in the string. A colon without a number before is an unknown flag that is ignored.
- flags: zero or more single characters that modify the behaviour of the conversion.
- width: decimal number denoting the minimum output field width. The output is unrestricted if the number is missing.
- precision: decimal number dennoting the digits after the decimal point when floating point numbers are converted; maximum field width else.
- Single termination character that selects the output format.

Flag characters may be absent or one or may of the following, but neither a decimal digit (except zero) or one of the conversion specifiers:

```
-   pad to the right, i.e. the result is left adjusted
+   use a plus sign for positive numbers (and zero)
0   pad with zeroes to the left, no padding to the right
p   pad with points in both directions
·   pad with Unicode '·' (&middot;) in both directions
"   Encloses in (double) quotes; doubling quotes inside, after truncation
'   group the result by "'" (see 'group' function)
_   group the result by "_" (see 'group' function)
t   technical exponential format (multiples of 3 only)
ø   use Unicode 'ø' for insignificant digits
₁₀  use Unicode '₁₀' instead of 'e' in exponential format
k   use SI postfix if possible instead of (small) exponents
K   use binary postfix ('ki','Mi' etc.) if applicable
… use the character '&hellip;' to mark truncation
```

The result of conflicting flags is undefined; no error is given. Any other character is silently ignored; which is also the case if it is not (yet) implemented.

The last character is the conversion specifier; one and only one of the following list is used to terminate the conversion specification:

```
s   string output; if not a string, convert by string catenation first
d,i an integer number is  to decimal
x,X convert integer number to hexadecimal
o   convert integer number to octal
b   convert integer number to binary
f   use fixed point format
e,E use exponential format
g,G use fixed format for small and expoential format for large ones
```

If the item to be converted is not an appropriate number, the conversion is treated as string conversion.

If a captial letter is used, all output characters are capital letters.

The source selector is not present in C and Python percent formatting. Selecting a source either explictely or implied that is not provided, siltently used void instead, which is represented as the string () here (in contrast to string catenation, where it is empty).

To print a number as percentile using the% modifier needs at least one other character after the leading percent:

```
format 'was %.%', 0.12
format 'was %', 0.12
```

13. 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.

Fields and Attributes

As the distinction between variable fields, control fields and attributes depends on the kind of item, the final processing is in the runtime.

Because attributes and control fields has precedence over variable fields, the runtime has first to look for the former, which may degrade field access.

Therefore, the compiler has a list of all attribute and control field names, and generates code to check these first, independent of kind. All other do not probe as attributes, but directly look for a field.

This also allows to mark newly introduced attributes as well as deprecated ones with compiler warnings.

Storage Management

This sections is partially outdated and needs update

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 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 offun2 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.

Remarks

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,

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 map maps integers to items. Integer 0 is reserved for void.

Rows and maps

Rows are kept as continuous memory of item pointer, re-allocated if necessary. Currently, the amount added is fixed to 50 items (same as initial), which seems rather suboptimal for large rows. But no real performance issues have been observed: Calculating 10 million primes, i.e. increasing the map 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.

Fields use a unidirectional linked list with the lastest used entry first which is quite efficient upto 100 entries. As fields require a name in the programme, there is no performance problem.

Small maps (with upto 32 keys currently) use a linked list like fields; if more are needed, a hash table is used, and linked lists still for collisions. If the number of keys exceeds the hash table size by a factor (currently 2), a larger hash table (factor 4 in size) is used. In order to avoid unexpected delays by re-hashing all keys, this is postponed until the key is actually used.

The map hash uses the default hash function.

When a map is released (use count dropped to zero), all elements are released too. The following test will show a performance issue in the case that a large map is filled and immediately relased again:

```
m =: new map
?# i =: from 1 upto 1'000'000
   m{i} =: i
m{i} =: ()
```

The last statement `m{i} =: ()` takes much longer (quadratically) than filling the map, which is not the case with a row, because the latter releases all elements in decreasing key order, while the map releases them more or less in random order (hash slot by hash slot).

The reason is the map of all items maintained to detect memory leaks, which is a relative small hash table with a fixed size, currently 4999 slots, and using linked lists for overflow, with the least created on top. In the above case, the average length is 200 elements. If these are released in creation order, each element is at the end of the chain. If a map is relased, the elements are in slot order, i.e. nearly random with higher probobality for last entered ones. Thus, the time is smaller, but still high.

Note that in normal operation, many items are relased soon after created, thus the LRU chains are quite efficient even for a large number of items used.

If not enabled, the above performance problem vanishes (to a linear time for relasing a map). Keeping track of all items in use proved to be very useful during the development of the runtime and binding libraries, but may in future be switched on only in debug mode.

There is an alternative implementation under development, where items are allocated in chained blocks of say 500, providing a function to error print all used items or to provide them as scan (not a row, may be to many). T

Tagged bunches

It might be an implementation option if class 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 TAV

For bootstrapping, a restricted version of TAV, *level 0 TAV*, 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.

TAV-PL 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 TAV functions. These may, but often will not be a 1:1 relation in order to better adopt the interface to the programming concepts of TAV-PL.

Practically always these binding interfaces need to use data structures different from the items used in TAV-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 TAV 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 xxxxxxx identifies the interface.

Also, a row or map can be used, tagged and locked. Note that C programs can set the lock count to maximum and thus inhibit unlocking.

Multiarch 32bit and 64bit

The compilation can be for both, 32- and 64-bit architectures; just adjust the parameter in the makefile. Except for a few places where old 32-bit interfaces to libraries are used, the resultant programmes should only differ in size and execution time.

If in a 64-bit operating system the necessary 32-bit libraries (i.e. *multiarch* in Debian Linux) are available, the 32-bit variant will be the default, because it requires less space and is often quicker. Note that integers use 64-bit words anyhow, so there is no performance penalty even for integer applications with larger numbers.

Normally, either the 32-bit or the 64-bit variant is installed, selected by the approriate parameter, and with one set of libraries only, which are not distinguished by name. Currently, the selection is in the tuf script, as this is copied during installation.

Alternatively, both versions can co-exist (on a suitable system); then, the additional parameter-m32 and -m64 can be used in the tuf script. In this case, both sets of libraries are present, preferrably in subdirectories TAV32 and TAV64. In this case, the tuf script checks for the subdirectories and uses the 32 version per default if no parameter is given.

Note that currently, the compilation is done by a script tuf that compiles and links a programme; compile-only requires the option -c, while compile-and-run is done by the option -r.

---

[1] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.98.965&rep=rep1&type=pdf.

[2] *PL* an abbreviation for *programming language*, not *Poland*

[3] meaning not equal, but with small differences that often break the program.

[4] https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/,

[5] Different symbols, e.g. tuf and -c, would clarify their special status, but would finally not be successful to win against the tradition

[6]

[7] Raymond T. Boute: *The Euclidean Definition of the Functions div and mod* ACM TOPLAS vol.14 no.2 April 1992, pp. 127-144.

[8] Also among the conditions is tuf, because he extends the criteria to real numbers. The condition -c had probably been left out to allow real numbers for the remainder.

[9] in contrast to AWK, where using the key for a map creates the cell

[10] Flow analysis could nevertheless give a compile time warning

[11] Actually, the last condition should read tuf or -c if the latter is supported by the compiler

[12] The first runtime uses such a construct to save space for short strings not exceeding 6 characters.

[13] like the lifeboats are removed once a ship is used daily

[14] If the fields are a chained list of name-value pairs, a second list for name-function ref pairs is used.

[15] except that is is not possible to extend the language later to allow -r for *equal or less*

[16]

[17]

[18]