TISBL language specification

Rob Mitchelmore

Revision history

2.0 First LaTeXversion. Massive expansion of the original, and rethink of some syntax.

1.0 The original HTML version from 2004.

1 Language overview

1.1 Introduction

TISBL is an extremely simple and very definitely unusable stack-based language influenced by FORTH, PostScript, LISP, TeX, and which punctuation characters looked pretty.

TISBL has no code blocks and no real idea of scope; nor are there any named functions or variables; it also has a somewhat blurred distinction between code and data.

Abandon hope, all ye who enter here.

1.2 Making the computer ignore you

Comments are introduced with a `%` sign and continue to the end of the line:

```
% everything from here to EOL is ignored
```

1.3 Simple operations

TISBL is a stack machine; each TISBL subprogram (more on this later; for the moment, you can assume that each program is a subprogram) has two data stacks and an execution stack. There are two basic data types in TISBL: word and number. A word is an arbitrary string of characters (although for syntactic reasons, a literal word cannot have whitespace within it); a number is either an integer or a floating-point number.

To push a word onto the stack, prefix it with a `'` (single quote). For example:

```
'hello % pushes the word "hello" onto the stack
```

To push an number, use the # character; for example:

```
#3 % pushes the number 3 onto the stack
```

Multiple tokens can be seperated with any amount of whitespace (being either the space character, tabs, or carriage returns and/or linefeeds). For example:

```
#3 #4

#3
#4
```

and `#3 #4`

are all equivalent, pushing 3, then 4 onto the stack.

To perform operations upon the stack, verbs are invoked. A verb consists of a backslash (\) followed by one or more characters that are not whitespace and not full stop (.), comma (,), colon (:) or semicolon (;). A verb pulls its input from the stack and pushes its results back to the stack. For example:

```
#3 #4 \+
```

pushes 3 and 4 onto the stack, adds them (\+ attempts to pop two arguments from the stack), and pushes 3+4 onto the stack. To output, use \out:

```
#3 #4 \+ \out
```

produces '7' on the screen, as expected.

The verbs available by default are listed in the standard library section of this document, below.

1.4 Using the second data stack

Each TISBL execution context has two data stacks; the primary one which has been used so far, and a second one, which is called '`:`' for no particularly good reason.

To push data onto a stack other than the main one, place the stack's name before the type marker, for example:

```
:#3
```

pushes 3 onto the second data stack. To invoke a verb, the verb must be given two stacks: one from which to read the input, which goes after the backslash but before the name, and one onto which to post the output, which goes after the name, for example:

```
\dup:
```

which copies an item from the top of the main data stack to the second data stack,

```
\:dup
```

which copies an item from the top of the second data stack to the main data stack,

```
\:swap:
```

which swaps the top two items of the secondary stack, and

```
:#3 :#4 \:+
```

which adds 3 and 4 from the secondary data stack and pushes the result onto the main data stack.

1.5 Novel uses of \rm: or, the execution stack

When a TISBL program is loaded, what is actually loaded from the text file is the initial state of the execution stack, each token being put onto the stack as a word, with the first token in the file as the head of the stack (this is to ensure some semblance of normal execution order). To execute the program, each word is popped off the execution stack and executed; thus the top of the execution stack is always the next token to be executed. Once loaded, and once the program is running, the execution stack becomes another stack accessible to the program; its name is '`,`'. For example:

```
\,rm
```

which skips the next token (removes the head of the execution stack),

```
\,dup,
```

which will make the next token execute twice, and on a more convoluted note:

```
:'#3 :'4 \:+:  \:dup,
```

which produces a word "#34" on the second data stack and copies that to the execution stack; the end result of this being to push 34 onto the main data stack. A more confusing, and rather crueller, variant on this would be:

```
:':# :#34 \:+ \dup,
```

this constructs a ":#34" word on the main data stack from the ":#" word and the 34 number on the second data stack, then copies that ":#34" from the main data stack onto the execution stack, thus causing the number 34 to be pushed to the second data stack as the next operation.

Phew.

1.6 Code blocks, \exec and \verb

Consider for a moment the following code:

```
''hello ''world '\+ '\out
```

After this, the primary data stack will consist of:

```
TOP => '\out, '\+, ''world, ''hello
```

If one then executes

```
#4 \multipop:
```

the second data stack will look like:

```
TOP => ''hello, ''world, '\+, '\out
```

Note that the order of the tokens has been effectively reversed in the transfer between stacks, as \multipop moves items one at a time. If instead of pushing onto the second data stack, however, one pushes onto the execution stack, then the tokens will be executed and "helloworld" will be output.

The \exec verb extends this concept by creating a new subprogram and doing a multipop onto its execution stack. A subprogram has its own primary and secondary data stacks, and things written to these stacks will not affect the world outside the subprogram. For example:

```
''aaa '#3 '#4 '\+ '\out
#5 \exec
```

will print, as expected, 7; and when the subprogram ends, the remaining item (`'aaa`) on its main data stack is discarded:

```
''aaa '#3 '#4 '\+ '\out
#5 \exec \out
```

will give a stack underflow error when it hits the \out.

There is also a facility in TISBL to add verbs to the language, and this is done with the \verb verb. \verb first pops a word from the stack, and then a code block as \exec does, and associates the word with the code block in the global 'verb pool'. When the verb is executed, the code block is executed in a new subprogram as if it was \exec-ed. For example:

```
'#3 '#4 '\+ '\out #4
'seven \verb
```

defines a verb that outputs "7" called "seven"; and calling this verb:

```
\seven
```

will execute that subprogram and output "7".

1.7 Talkative Subprograms

So far, the subprograms examined have had a major shortcoming: they cannot take parameters and push back results. This shortcoming is addressed by the fact that each subprogram has a reference to an input stack and an ouput stack, much as the built-in verbs have. A subprogram cannot write to its input stack, and cannot read from its output stack; these stacks are both referred to as '.', and the correct stack is used depending on context. For example:

```
.'hello
```

pushes the word "hello" onto the current subprogram's output stack, while

```
\.+
```

pops the two arguments for + from the current subprogram's input stack and pushes the result onto the primary data stack; and

```
\.dup.
```

which duplicates the top of the input stack onto the top of the output stack.

When the subprogram is executed, the input and output stacks of the block are filled in from the verb that caused the execution; in the case of \exec, the input and output stacks passed to \exec, and in the case of \verbed code blocks, the input and output stacks passed to the verb name. For example:

```
#3.5
'\.mv '#2 '\* '\dup.  #4 \exec:
```

puts #7.0 onto the second data stack (the code and the input were from the main stack as passed as the input stack to \exec; the output goes to ':' as that is the output stack to \exec). Another example:

```
'\.mv '#2 '\* '\dup.  #4 'timestwo \verb
#3.5 \timestwo:
```

is equivalent to the \exec example above. More interesting things can be done with verbs, however:

```
'\.timestwo '\timestwo.  #2 'timesfour \verb
```

which defines a timesfour verb in term of timestwo, using the '.' notation for input and output stacks.

1.8 \if and \while

\if implements conditionals, and works as follows: First, it pops a code block from its input stack as \exec does; then it pops a single item from its input stack. If that item is not 0 then it executes the block as a subprogram as if it had been run with \exec, inheriting its input and output stacks from the \if verb. Otherwise, the code block is discarded. An example:

```
'this \_
#1 '.'will #1 \if
#0 '.'won't #1 \if
\+ \_ 'be \_ \+ 'executed.  \+ \out
```

which will always produce "this will be executed."; as the #0 in the second \if will prevent that code block from being executed.

The \while construct is a variant on this. It too first pops a code block from its input stack, then it pops an item and if it isn't 0, executes the code block. While \if stops there, however, \while continues: when the code block has finished executing, it pops another item, and if it isn't 0, executes the code block again in a new subprogram (data stacks are not persistent between iterations in a \while - it works much like repeated calls to \exec on an identical code block); and it continues thus until it pops a 0, whereupon it stops and allows the program to continue. An example is:

```
#0 #1 #2 #3 #4
'\.dup '\out #2 \while
```

which produces the output

```
3210
```

The #4 is never printed, as the \while swallows the first item remaining on the stack each iteration. By making the input and output stacks of \while the same, it is possible to implement something akin to the while loop in other languages, by pushing something back onto the output stack at the end of the loop:

```
#100 #1
'\.mv '#1 '\- '\dup.  '\dup.  '\out #6 \while
```

which produces the numbers from 100 to 0.

This example is slightly more convoluted than previous ones; so, a dissection. The 100 is the number to count down from; the #1 is the initial loop condition. This item must be present as \while takes one item before evaluating the code block for the first time. Thus, when the code block executes for the first time, there is only the 100 left on the stack. The code block then moves that 100 to its own local stack, subtracts 1 from it, duplicates it twice back onto the parent

stack (remember, \while will swallow one item, and one needs to remain on the parent stack as the datum for the next iteration). It then outputs the number and ends. The \while then pops the top of its stack, finds that it is not 0, and iterates again.

1.9 Messing With the Future, Part 2

In TISBL, subprograms have the ability to mess with their parent context's execution stack (thus removing its single sole possible remaining purpose, that of a fairly secure programming language). The parent's execution stack is called ';', in a move designed to annoy Java, C, and Pascal programmers who are used to putting a semicolon at the end of each (or most) lines, as this way there will be no syntax error, but the program is likely to do something violently unpredictable.

Note that while the subprogram is running, its parent is frozen; thus, any instructions pushed onto ; will not be executed until the subprogram has finished executing. For example:

```
''hello '\_ '\out ';'\world #4 'hello \verb
''world '\n '\out #3 'world \verb
\hello
```

would output

```
hello world
```

as the "hello" verb forces the program to execute "\world" following it by pushing the "\world" word onto its caller's execution stack. Another, even more useless example:

```
''hello '\_ '\out ';'\hello #4 'hello \verb \hello
```

which outputs "hello hello hello hello ..." until the user gets bored and kills the interpreter[1].

1.10 A note on syntax checking

It is important to note that a TISBL interpreter should not check each token for correct formatting as it is loaded, but only as it is executed; for example:

```
'hello \_ \out /error 'world
```

should not bail out until execution of the token "/error" is attempted.

[1] On interpreters which buffer their output, like the THINK Pascal one, you may see no output at all.

2 Language spec

This outlines an interpreter for TISBL. Algorithms are illustrative only and should be viewed with appropriate levels of critical thinking. Equivalent or entertainingly non-equivalent implementations are still conforming for what little that is worth.

2.1 Interpreter state

The interpreter state consists of a stack of *contexts*. When the program starts the state contains one context.

Each context consists of three stacks:

- A primary stack.
- A secondary stack.
- An execution stack.

It also consists of three references to other stacks:

- Its input stack
- Its output stack
- Its parent's execution stack

Implementations may limit the depth of the context stack. The interpreter state also contains a single, global verb table. Each entry in this table consists of a verb name and the initial execution stack for a context.

2.2 Stacks

A stack must provide three primitive operations:

Push which pushes an item onto the stack

Pop which removes the top item from the stack and returns it.

Peek which returns the top item from the stack without removing it.

Each item in the stack may be:

- A string
- An integer
- A floating-point number

Integers should be at least 16 bits wide. What kind of float is used is implementation-specific. Strings may be either Unicode, Latin-1 or ASCII. If implementations really want to use EBCDIC I suppose they may.

2.3 Running a context

To run a context:

- While the execution stack is not empty:
  - Pop an item from the execution stack
  - Parse it
  - Execute the parsed representation

2.4 Parsing a token

A token consists of

- What kind of token it is: number, word or verb.
- The name of its input stack, if it is a verb.
- The name of its output stack
- Its contents.

In order to parse a token:

- Is the first character a backslash (\)?
- If so:
  - Chop the backslash off
  - Attempt to read a stack character (see below).
  - If you get one, store a reference to the stack it represents in the token's input stack field and chop the character off.
  - If you don't, store a reference to the default data stack in the token's input stack field.
  - Look at the last character; if it is a valid stack character, then read it and store a reference to the stack it represents in the token's output stack field. Chop off the stack character.
  - What remains is the contents of the token.
- Otherwise: (the first character is not a backslash)
  - Attempt to read a stack character (see below)
  - If you get one, store a reference to the stack it represents in the token's output stack field and chop the character off.
  - Attempt to read a hash (#) or a single quote ('). If you get the former, store that the token is a number. If the latter, store that it is a word. If you get neither, bail out.
  - The remainder of the string is the contents of the token.

A stack character is one of a full stop (.), a colon (:), a comma (,) or a semicolon (;).

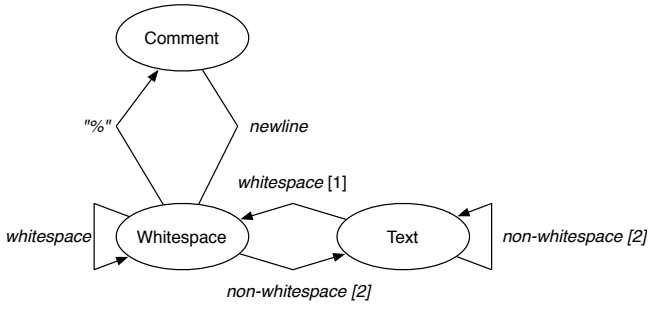
2.5 Executing a token

To execute a token:

- If the token is a word, then push the contents of the token onto the token's specified output stack as a string item.
- If the token is a number then:
  - If the contents of the token contains a decimal point (.), convert the contents into an implementation-dependent float format and push that onto the token's specified output stack.
  - Otherwise, convert the contents of the token into an integer and push that onto the token's specified input stack.
  - Should the content of the token not convert to being a number, bail out.
- If the token is a verb, then:
  - If the content of the verb matches the name of a verb in the verb table (*case-sensitive!*) then
    - Start a new context
    - Set the contents of the execution stack of the new verb to be that matching the verb name in the verb table (*preserving ordering*)
    - Set the input and output stacks of the context according to the input and output stacks of the token.
    - Execute the new context (see above)
    - Clean up context as necessary.
  - Otherwise, if the content of the verb matches the name of a standard library function, run it
  - Otherwise, bail out.

2.6 Loading a file

A file consists of a set of whitespace-separated tokens. Comments begin with a percent sign (%) either at the beginning of a line or after a whitespace and continue until the end of the line. A state machine for this is:



Notes [referring to the bits in square brackets]

1. If the buffer that you're using to assemble the token is not empty, put its contents onto the *bottom* of the initial execution stack then clear the buffer.

2. Append the character to the buffer

To load a file:

- Create an initial context with invalid input and output stacks.
- Parse the input file according to the above state machine.
- Execute the initial context.

3 Standard Library

3.1 \trace=0 and \trace=1

Should set a bit of internal state for debug; when trace=1 the interpreter should print the contents of the primary, secondary and execution stacks before every token is executed.

3.2 \exec

\exec creates a new context with the same input and output stacks as itself; it pops an integer from its input stack, then pops that number of elements from its input stack, pushing each one onto the execution stack of the new context.

$exec(input, output) \rightarrow$
  $ctx \leftarrow$ new Context
  $ctx.input \leftarrow input$
  $ctx.output \leftarrow output$
  $count \leftarrow pop(input)$
  die unless $count$ is a integer
  for i in $[1..count]$ do
    $item \leftarrow pop(input)$
    $push(ctx.exec, item)$
  execute ctx

3.3 \verb

\verb creates a new verb in the verb table. First, it creates a new stack. It pops a string from its input stack to be the name of the verb and stores it, then pops an integer from its input stack. It pops this number of elements from its input stack, pushing them onto the new stack as it goes. It then adds a mapping from the name to the new stack to the verb table.

verb(*input*, *output*) →
  s ← new Stack
  name ← *pop*(*input*)
  die unless *name* is a string
  count ← *pop*(*input*)
  die unless *count* is an integer
  for *i* in [1..*count*] do
    item ← *pop*(*input*)
    push(*s*, *item*)
  add (*name*, *s*) to verb table

3.4 \if

\if conditionally executes code. First, it creates a new context with the input and output stacks set the same as those on the verb. Then, it pops an integer from its input stack, and pops that number of elements from its input stack, pushing each onto the execution stack of the new context as it goes. It then pops an element from its input stack; if that element is a number and 0, then it discards the new context. Otherwise, it executes it.

if(*input*, *output*) →
  ctx ← new Context
  ctx.input ← *input*
  ctx.output ← *output*
  count ← *pop*(*input*)
  die if *count* is not an integer
  for *i* in [1..*count*] do
    item ← *pop*(*input*)
    push(*ctx.exec*, *item*)
  condition ← *pop*(*input*)
  if *condition* is integer or float and *condition* = 0 then
    discard *ctx*
  else
    execute *ctx*

3.5 \while

\while loops over a piece of code. First, it creates a new stack. It pops an integer from its input stack, and pops that number of elements from its input stack, pushing each onto the new stack as it goes. It then iterates: on each iteration, it pops an element from its input stack. If this number is a number and 0 then it terminates; otherwise, it creates a new context, clones the new stack into the execution stack of the new context, and runs the context.

while(*input*, *output*) →
  s ← new Stack

$count \leftarrow pop(input)$
die unless $count$ is an integer
for i in $[1..count]$ do
  $item \leftarrow pop(input)$
  $push(s, item)$
repeat
  $condition \leftarrow pop(input)$
  if $condition$ is integer or float and $condition = 0$ then
    $endloop \leftarrow 1$
  else
    $ctx \leftarrow$ new Context
    $ctx.input \leftarrow input$
    $ctx.output \leftarrow output$
    $ctx.exec \leftarrow$ clone of s
    execute ctx
until $endloop = 1$

3.6 \not

\not pops an element from its input stack; if it is numeric and zero then push 1 onto its output stack; otherwise push 0.

$not(input, output) \rightarrow$
  $e \leftarrow pop(input)$
  if e is an integer or a float and $e = 0$
    $pushInteger(output, 1)$
  else
    $pushInteger(output, 0)$

3.7 \swap

\swap pops the top two elements of its input stack and pushes them swapped on its output stack.

$swap(input, output) \rightarrow$
  $a \leftarrow pop(input)$
  $b \leftarrow pop(input)$
  $push(output, a)$
  $push(output, b)$

3.8 \dup

\dup duplicates the first element of its input stack onto its output stack.

$dup(input, output) \rightarrow$
  $e \leftarrow peek(input)$
  $push(output, e)$

3.9 \rm

\rm removes the top element of its input stack.

$rm(input, output) \rightarrow$
  $pop(input)$

3.10 \mv

\mv moves the top element of its input stack to its output stack

$mv(input, output) \rightarrow$
  $e \leftarrow pop(input)$
  $push(output, e)$

3.11 \multipop

\multipop pops a number from its input stack; it then goes through that number of cycles of popping an item from the input stack and popping it onto the output stack.

$multipop(input, output) \rightarrow$
  $count \leftarrow pop(input)$
  die unless *count* is an integer
  for i in $[1..count]$ do
    $e \leftarrow pop(input)$
    $push(output, e)$

3.12 \+

\+, part one of the unnecessarily complicated arithmetic library, adds two stack elements. If both are numbers it adds them and pushes the result back on the stack; if they are both strings then the string on the top of the stack is concatenated to the one beneath it.

$+(input, output) \rightarrow$
  $sec \leftarrow pop(input)$
  $fst \leftarrow pop(input)$
  if fst is string or sec is string
    $sec_s \leftarrow toString(sec)$
    $fst_s \leftarrow toString(fst)$
    $push(output, concat(fst_s, sec_s))$
  else if fst is a float or sec is a float
    $sec_f \leftarrow toFloat(sec)$
    $fst_f \leftarrow toFloat(fst)$
    $push(output, sec_f + fst_f)$
  otherwise, both must be integers
    $push(output, sec + fst)$

3.13 \−

\− subtracts two stack elements. It pops its second element then its first (so that the order in which it operates on its operands is the same as they appear in the source file). If both are numbers, then subtract the second from the first. If one is a number and one is a string, then remove that number of characters off the end of the string, rounding if necessary; if both are strings, remove all characters in the second string from the first.

$-(input, output) \rightarrow$
  $sec \leftarrow pop(input)$
  $fst \leftarrow pop(input)$
  if fst is an integer and sec is an integer
    $push(output, fst - sec)$
  else if fst is an integer and sec is a string
    $push(output, sub_{(s,i)}(sec, fst))$
  else if fst is a string and sec is an integer
    $push(output, sub_{(s,i)}(fst, sec))$
  else if fst is an float and sec is a string
    $fst_i \leftarrow toInt(fst)$
    $push(output, sub_{(s,i)}(sec, fst_i))$
  else if fst is a string and sec is an float
    $sec_i \leftarrow toInt(sec)$
    $push(output, sub_{(s,i)}(fst, sec))$
  else if fst is a string and sec is a string
    $push(output, sub_{(s,s)}(fst, sec))$
  else if fst is a float or sec is a float
    $sec_f \leftarrow toFloat(sec)$
    $fst_f \leftarrow toFloat(fst)$
    $push(output, fst_f - sec_f)$

$sub_{(s,i)}(str, int) \rightarrow$
  return leftmost $length(str) - i$ characters of str

$sub_{(s,s)}(haystack, needle) \rightarrow$
  $ret \leftarrow$ ""
  for each character c in $haystack$, in order
    if c is not in $needle$
      $ret \leftarrow ret + c$
  return ret

3.14 *

* is a multiplication operator. It pops its second operator followed by its first. If both are numbers, then it multiplies them. If one is a string and one is a number, then it repeats that string that number of times. If both are strings,

then it replaces every occurrence of the first character of the second string in the first string with the whole second string.

$*(input, output) \rightarrow$
  $sec \leftarrow pop(input)$
  $fst \leftarrow pop(input)$
  if fst is an integer and sec is an integer then
    $push(output, fst * sec)$
  else if fst is a string and sec is an integer then
    $sec_f \rightarrow toFloat(sec)$
    $push(output, mul_{(s,f)}(fst, sec_f))$
  else if fst is a string and sec is a float then
    $push(output, mul_{(s,f)}(fst, sec))$
  else if fst is a float and sec is a string then
    $fst_f \rightarrow toFloat(fst)$
    $push(output, mul_{(s,f)}(sec, fst_f))$
  else if fst is an integer and sec is a string then
    $push(output, mul_{(s,f)}(sec, fst))$
  else if fst is a string and sec is a string
    $push(output, mul_{(s,s)}(fst, sec))$
  else, one is a float
    $sec_f \rightarrow toFloat(sec)$
    $fst_f \rightarrow toFloat(fst)$
    $push(output, fst_f * sec_f)$

$mul_{(s,f)}(str, iter) \rightarrow$
  $count \rightarrow iter$
  $buffer \rightarrow$ ""
  while $count \geq 1$
    $buffer \rightarrow concat(buffer, str)$
    $count \rightarrow count - 1$
  if $count > (1/length(str))$
    $chars = round(length(str) * count)$
    $buffer \rightarrow concat(buffer,$ leftmost $chars$ characters of $str)$
  return $buffer$

$mul_{(s,s)}(str, expand) \rightarrow$
  $buffer \rightarrow$ ""
  for each character c in str, in order
    if $c =$ first character of $expand$
      $buffer \rightarrow concat(buffer, expand)$
    else
      $buffer \rightarrow concat(buffer, c)$
  return $buffer$

3.15 \div

\div divides two stack elements by one another. It pops first its second argument, then its first: if both are numbers, then it divides the first by the second and pushes the result. If one is a string and one a number, then it divides the length of the string by the number and returns that number of characters from the left of the string. If both are strings, then it replaces in its first parameter every occurrence of its second parameter with the first letter of its second parameter

$div(input, output) \rightarrow$
  $sec \leftarrow pop(input)$
  $fst \leftarrow pop(input)$
  if fst is an integer and sec is an integer then
    $push(output, fst/sec)$
  else if fst is a string and sec is an integer then
    $sec_f \rightarrow toFloat(sec)$
    $push(output, div_{(s,f)}(fst, sec_f))$
  else if fst is a string and sec is a float then
    $push(output, div_{(s,f)}(fst, sec))$
  else if fst is a float and sec is a string then
    $fst_f \rightarrow toFloat(fst)$
    $push(output, div_{(s,f)}(sec, fst_f))$
  else if fst is an integer and sec is a string then
    $push(output, div_{(s,f)}(sec, fst))$
  else if fst is a string and sec is a string
    $push(output, div_{(s,s)}(fst, sec))$
  else, one is a float
    $sec_f \rightarrow toFloat(sec)$
    $fst_f \rightarrow toFloat(fst)$
    $push(output, fst_f/sec_f)$

$div_{(s,f)}(str, num) \rightarrow$
  $chars \leftarrow round(length(str)/num)$
  return leftmost $chars$ characters of str

$div_{(s,s)}(str, divby) \rightarrow$
  $buf \leftarrow str$
  while $divby$ occurs in buf
    $idx \leftarrow$ index of first character of $divby$ in buf
    delete $length(divby) - 1$ characters from buf starting at $idx + 1$
  return buf

3.16 \n

\n concatenates a newline onto the end of the string representation of the top of the stack.

$n(input, output) \rightarrow$
  $e \leftarrow pop(input)$
  $s \leftarrow toString(e)$
  $push(output, concat(s, \texttt{"\textbackslash n"})$

3.17 _

_ concatenates a space onto the end of the string representation of the top of the stack.

$\_(input, output) \rightarrow$
  $e \leftarrow pop(input)$
  $s \leftarrow toString(e)$
  $push(output, concat(s, \texttt{"␣"})$

3.18 \word?

\word? pops a stack element, and pushes 1 if it is a string, and 0 if it isn't.

$word?(input, output) \rightarrow$
  $e \leftarrow pop(input)$
  if e is a string
    $push(output, 1)$
  else
    $push(output, 0)$

3.19 \number?

\number? pops a stack element, and pushes 1 if it is a number, and 0 if it isn't.

$number?(input, output) \rightarrow$
  $e \leftarrow pop(input)$
  if e is an integer or a float
    $push(output, 1)$
  else
    $push(output, 0)$

3.20 \integer?

\integer? pops a stack element, and pushes 1 if it is a number, and 0 if it isn't.

$integer?(input, output) \rightarrow$
  $e \leftarrow pop(input)$
  if e is an integer
    $push(output, 1)$
  else
    $push(output, 0)$

3.21 \float?

\float? pops a stack element, and pushes 1 if it is a number, and 0 if it isn't.

$float?(input, output) \rightarrow$
  $e \leftarrow pop(input)$
  if e is a float
    $push(output, 1)$
  else
    $push(output, 0)$

3.22 \die

\die makes the interpreter die.

$die(input, output) \rightarrow$
  bail out

3.23 \out

\out pops an element, converts it to a string, and sends it out to standard output or equivalent. Note that it should not introduce any extra spaces: numbers should not be space-padded fore or aft.

$out(input, output) \rightarrow$
  $e \leftarrow pop(input)$
  $print(toString(e))$

3.24 \in

\in reads a line from standard input and pushes it onto its output stack as a string.

$in(input, output) \rightarrow$
  $str \leftarrow readLine$
  $push(output, str)$

4 Things for immediate discussion

4.1 Support predicates

It might be a good idea to add a way of querying support for language extensions. The proposed way of doing this is to introduce a new token type, called support predicates. The distinguishing character for these tokens is a question mark.

When one of these tokens is executed, the interpreter checks whether the content of the token is one that it recognises. If it is, then the interpreter pushes 1 onto the token's output stack; otherwise, it pushes 0. The interpreter is absolutely required to push an integer on the output stack for *every* support predicate it encounters.

Example:

```
% bail out if no graphics support.
?graphics ''NoGraphicsSupport '\out '\die #3 \if
'red \colour #10 #10 #30 #30 \rectangle
```

4.2 \let

\let would *copy* the definition of a verb, be it a verb in the standard library or one that has been previously defined. Effectively, it would allow for renaming of verbs.

Example:

```
% Ungrammatical and unhelpful!
% Redefines 'lest' to be the same as 'if'
% Then redefines 'if' not to work any more.
'lest 'if \let
''UnhelpfulComment '\out #2 'if \verb
#1 ''PrintThis '\out #2 \lest
```

4.3 Explicit \word, \number

Since \in produces only string stack items, it'd be useful to be able to turn those into numbers.

Example:

```
% Input a number, add 1 to it, output that
\in \number #1 \+
'OneMoreIs \_ \swap \+ \out
```

Possibly these should be marked with a `!` on the end — `\number!`, `\word!`, `\float!`, `\integer!` and `\string!`?