SOPL

1. Introduction and Motivation

SOPL is an imperative programming language with some peculiarities that set it aside from ‘normal’ languages. You may regard it as an ‘esoteric’ and/or ‘weird’ language. (My personal aim rather is to construct a ‘beautiful’ language - this of course means different things to different people). In any case it is not meant for productive use but rather an experiment. You should regard programming in SOPL as a kind of brain-teaser. Let’s see how it works and have fun using it.

Keep in mind that the language is interpreted, so execution tends to be slow.

Motivation:
Due to its construction, the language encourages you to find unusual or non-standard solutions to old (and new) problems.

When beginning to tackle a problem with SOPL I often had the feeling that the language is missing some features and the problem isn’t solvable. After some reasoning however, it turned out that the features are sufficient and you can achieve your goal.

You should look at the code samples in the directories ‘samples’ and ‘features’ to get a notion of how things can be done.

2. Basics

Regarding the program structure, SOPL tries to borrow a few ideas from natural languages:

a) Sentences: Looking at natural languages and their sentence structure, it turns out that the majority of languages has an SOV structure (Subject - Object - Verb), that is, the verb comes at the end of the sentence. We regard an SOPL program as a sequence of sentences where each sentence ends with a verb which determines the kind of action taken.

Note verbs are only regarded as such if they are in sentence final position, otherwise they are simply regarded as literals.

b) References: A sentence can refer to other sentences by relative references like ‘this’ (meaning the result of the sentence just before the current one), ‘that’ etc.

Note The original version of SOPL only supplied these relative references. It turned out, however, that this was the greatest source of programming errors. The current version of SOPL therefore allows sentences to be named so that they can be referenced by their names. (This requires the flag -ext (extension) to be set when calling the interpreter). You may not use the same name more than once. If you want to, you can therefore regard these names as immutable variables.

c) The basic meaning of a verb or reference may be changed by using certain suffixes.

d) conditions are realized by a kind of bulleted lists

e) The number of parameters to a verb is usually arbitrary, but organized in at most 4 different groups called ‘do list’, ‘for list’ ‘use list’ and ‘with list’.

3. General Rules and Reasonings

3.1 Rules

- Note that SOPL is case sensitive. Thus ‘Add’ is different from ‘add’ or ‘adD’.
- all lists are flat, i.e. there is no substructure for a list (other than using grouping words)
- Though we feel that a certain amount of syntactic sugar is ok, we try to keep the set of verbs small. Verbs may therefore be eliminated in subsequent versions of SOPL, if there is no (more) need for them.
- We try to restrict the complexity of the language to at most 30 verbs and 12 modifiers.
- There are no explicit loops (other than -perRow and -combine) in SOPL. You should use recursion.
- A program is made up of a sequence of expressions which may refer to each other.

4. Hints and Decisions

4.1 Header

I could have introduced a ‘main’ paragraph as the first paragraph but I choose an unnamed header for the following reasons:

- It is not called by any other part of the program explicitly, so a nameless header seems more appropriate.
- There are special verbs (so far only ‘include’) that may only be used in the header. Therefore it should not be confused with an ordinary paragraph. Future releases may introduce more header verbs. Include itself is usually not regarded as part of the program code but rather as a reference to other programs. As such, it is usually not regarded as ‘ordinary’ code.

4.2 Override

Though you may choose any name as a paragraph name, it is not possible to override standard verb definitions. If you do so, the parser will parse these definitions like any other paragraph but they will never be called. The standard verb will be executed instead.

4.3 Error handling

There are only few error handling mechanisms in this language and I do not currently intend to extend them. Most errors arise from wrong or insufficient data supplied to a verb. If necessary this should be handled by checking the data.

4.4 OS access

There are no methods to access OS functions and I have no plans to introduce these. After all, I regard the language as esoteric and not as a general problem solver. So I currently feel no need to implement any kind of OS access.

4.5 Calling one program from another

There is no need for a special verb or any other feature to achieve this, because it can be done by including the other program in question and simply calling their main paragraph (the one called in the header).

4.6 Performance

I tried to improve performance be rewriting the code using pointers where it seemed appropriate. This, however, did not improve performance.

5. Program structure

A SOPL program (a file with extension .sopl) consists of a header followed by a body. While both parts are made of sentences (which in turn are sequences of words), the body part is subdivided into one or more paragraphs

Header:

- the header may optionally start with one or more include file definitions. These are file lists (with their path given relative to the path of this file) terminated by the verb include. Note that so far spaces cannot be part of a file name.
- the header concludes by a call of a paragraph (defined in the body below).
- the header must be terminated by at least one blank line
- The body begins after the blank line that terminates the header and ends at the end of the file
- the body is made up of paragraphs, consisting of a sequence of sentences and terminated by (at least) one blank line.
- sentences are sequences of words terminated by a dot.

For examples refer to the samples directory.

6. Program call

A SOPL program is executed by calling the interpreter and supplying the program file as a parameter. The format of a call is:

```
SOPL.exe <options> <sopl file>
```

Option	Explanation
-r	just read the sopl file and echo it back
-l	do a lexical analysis of the file and output its result
-p	do a parse of the file and output the result
-x	execute file (by interpreter)
-dxyz	set logging level to debug and specify debug types xyz, see below
-i	set logging level to info
-a	set logging level to extended warnings
-w	set logging level to warnings
-e	set logging level to error
-f	set logging level to fatal error
-n	do not write log output to the console (only to log file)
-t	set logging level to total (log numeric calculations)
-L	write all (regular) output to the log file
-std	standard version (for now this is the default, so you need not specify it)
-ext	extended version

Internal debug types

Option	Explanation
L	write debug output for Lexer
P	write debug output for Parser
I	write debug output for Interpreter
b	write debug output for plist
c	write debug output for paragraph context
i	write debug output for Item (during parse)
j	write debug output for jump calculation
r	write debug output for references
s	write debug output for sentence execution
t	write timing output for loops (perRow, combine, plist) [1])
x	write debug output for indexes
y	write debug output for setting stop level

[1]) note that this doesn’t work for nested loops yet

7. Paragraph structure

A paragraph

- must begin with a word followed by the symbol ':' and a line break. This word (without the colon) is the name of the paragraph.
- It is not allowed to give a paragraph more than one name.
- Paragraph names must be unique within the program
- Paragraphs are verbs and must be called like all other verbs explicitly by the paragraph-name followed by a dot to be run. You can view a paragraph as a function definition if you want to.
- The end of a paragraph is denoted by at least one blank line. To properly end the last paragraph of a file it must end with at least one blank line
- Note that with most editors, this means that it seems as if you have to supply two empty lines at the end of the file

8. Sentence structure

A sentence is a sequence of words ending with a verb. The words preceding the verb are regarded as parameters to the verb.

Exception:
A comment sentence is a line beginning with the symbols '//' and ending at the end of a line. If two consecutive slashes appear within a line preceded by a space, then the rest of the line is also regarded as a comment.

Each sentence (except for comment sentences/phrases) has a value, that is the result of the operation executed. The nature of the operation is given by the verb and explained in a separate chapter.
Certain verbs only accept certain types of parameters. These restrictions are explained with the verb definition (see below).

9. Word structure

To understand word structure we first define all the characters, that cannot be part of a word. These are called separator characters and special characters.

9.1 separator characters

Separator characters are:
space tab CR (carriage return) LF (line feed)

No other characters can be used to separate words, thus the following sequences are just one word:
5+7
1*2
(4-5)*2

9.2 other special characters

Special characters are:
:) |

these are used to mark a word as a paragraph name ':', a conditional label ')' and are not regarded as being part of a word.
|..| is used to return the size of a value reference instead of the reference itself.

9.3 words

Words cannot contain separator characters and must be separated by a separator or special character.

a word may be any of the following

- a name, that is a sequence of characters not containing separator characters
- a place holder, this is a # or a word starting with #. Not all verbs interpret place holders, see below
- a number, that is a sequence of digits which may contain at most one '.'. The dot however, must not be the first or last character of the sequence.
  Examples of valid numbers:
  0 007 107 019 100091.0 0.7 0.0
  Examples of invalid numbers:

0. .0 1.1. 0.2.2 300,000

- a literal, that is an arbitrary sequence of characters enclosed in “ and not containing any of the following characters:
  “, CR, LF, tab, page break. Literals may contain the following escape sequences: \t \n with their usual meanings and also \q meaning a double quote. Note, that these escaped characters only appear in console output

9.4 names

a name may be any of the following

- a verb, see chapter 11
- a paragraph name, see chapter 7
- a reference, see chapter 12
- a name of a group (parameter sequence), see chapter 10

9.5 place holders

These are described together with the verb or verb modifier they may be used with. So far, the following features may use place holders:

Name	Type	see Chapter
-perRow	verb modifier	11.6
-combine	verb modifier	11.5
join	verb	11.2
split	verb	11.2

10. Parameter sequences

The Parameter list may be subdivided into several sub lists by special key words (markers) which separate parameters into several sub lists also called groups.

They are only used to divide a list of parameters into sub lists but do not act as verbs. Note that the marker word is given **after** the list that it separates from the rest of the sentence. The list that is not terminated by a marker word but by the verb itself is called 'do list'.

parameter sequence	**Explanation**
<pl1> for <pl2>	subdivides the parameter list (<pln>) into a for list and a do list. The meaning of the lists depends on the verb used.
<pl1> use <pl2>	subdivides the parameter list (<pln>) into a use list and a do list. The meaning of the lists depends on the verb and the modifier used.
<pl1> with <pl2>	subdivides the parameter list (<pln>) into a with list and a do list. The meaning of the groups depends on the verb used.
<pl1> use <pl2> for <pl3>	subdivides the parameter list (<pln>) into a use list, a for list and a do list. The meaning of the lists depends on the verb and the modifier used.

Note that the words 'for', 'use' and 'with' are always interpreted as markers, regardless of the nature of the verb that ends the sentence. Also, the order of these words is free, you may write something like x with y for z use a <verb>

11. Verbs

All verbs must be terminated by a dot '.'. It must immediately follow the verb (stem) if the verb has no modifiers, otherwise it must immediately follow the last modifier.
The following lists give the explanation for all verbs of the language. It is to be understood that the parameters to the action are made up of the contents of the parameter lists, see above.

11.1 Header Verbs

Verb	**allowed parameter sequence**	**Explanation**
include	do	a list of sopl files which may be referenced in this file

11.2 Paragraph Verbs

the result of each verb action is always placed in the result list:

Verb	**allowed parame-ters**	**Exten-sion?**	**Explanation**
between	[use],for, do	no	insert all parameters of the for group between two adjacent parameters of the do group. if a use group is present, its first element gives the distance between two adjacent inserts. Thus '2 use' inserts after the second, forth,... do parameter.
del	for, do	no	delete the indexes provided in the for parameters from the do parameters. **Note that indexing is one-based**
expand	do	no	expand any word into a list of its characters. Note that spaces are ignored

find	for, do	no	for each value of the for parameters find the index in the do parameters (if present) and place it in the result list. **Note that indexing is one-based. The for parameter list should be unique.**
file	for, do	no	the for parameters are the operations that are executed for the files given as the do parameters, see below
freq	for, do	yes	the do list must have the form of a frequencies list = pairs of (keys,occurrences). The for list parameter determines the action taken: unify = make list unique with respect to keys most = return only most frequent pair(s), supposes that the list was unified. least = return only least frequent pair(s), supposes that the list was unified.
get	for, do	no	the for parameters contain a list of indexes. select the values corresponding to the indexes from the do parameters and place their value into the result list. **Note that indexing is one-based. The for parameter list may contain duplicate entries.**
id	do	no	place all input parameters into the result list
input	do	no	write the parameters as a prompt followed by ?, read a line from the console as input and return its value
ins	for, do	no	the first parameter of the for list is the index after which all subsequent for list parameters are inserted into the do parameter list. To insert at the beginning use 0 as the index. **Note that indexing is one-based**
join	[for], do	no	concatenate all parameters of the do group by the separator defined by the for group into one. If there is no for group then concatenate without a separator. The word #space in the

			for group may be used to define the separator space
mask	for, do	no	mask all parameters of the do group by the corresponding number defined by the for group and return the new list in the following way: the for parameter number gives the number of occurrences of the do parameter in the result. If the for parameter list is shorter than the do parameter list, it is repeated.
minus	for, do	no	determine the set difference between the for list and the do list (for - do)
nop	ignored	no	do nothing. This means that the result list content also isn't changed. Should only be used internally
output	do	no	the parameters are printed "as is" to the console in the sequence in which they are listed
pexpr	do	no	evaluates an expression using postfix notation. This is a stripped down version of plist, see there.
plist	[with],[use],for, do	no	see own chapter
print	do	no	the parameters are printed to the console in the sequence in which they are listed. Escape sequences are treated as such and enclosing double quotes omitted.
range	do: a list of 1 - 3 (integer) pairs	no	one number pair <n> <m> is expanded to a list of numbers n n+1 n+2 ... m or n n-1 n-2 ...m (if m < n). Two pairs <n1> <m1> <n2> <m2> produce a list like so: n1 n2 n1 n2+1 ....m1 m2 You may specify at most three such pairs as ascending or descending ranges. The second or third pair may contain the special characters * (=index of prev. pair) + (=index of prev. pair + 1) or - (=index of prev. pair -

			1)
readLines	[for,]do	no	reads the file and produces as a result a list of lines. You may additionally specify some characteristics with a forlist: keepEmpty= an empty line is output as the list element "<empty>" keepEol = every line is followed by the additional list element "<eol>" keepEof = at the end of the file, the string "<eof>" is added as a list element
returnVal-ue	do	no	add the list entries to the end of the return value list of the current paragraph. Note that on entering a new paragraph, the return value list is set to empty.
reverse	do	no	reverse the list of input parameters
set	for, do	no	the for parameter list must contain pairs of indexes and values. The corresponding lists elements (by index) in the do parameter list are then replaced by the given values. The changed list is returned. **Note that indexing is one-based**
sort	[for], do	no	sort the list of parameters meaning of for parameters, if present: desc = sort descending rows= sort by rows, expects 2 integers following: 1.) index to sort 2.) number of cols per row
split	[for], do	no	split all Parameters of the do group by the separator defined by the for group and return the new list. If the for parameter contains more than one character, only the first is used. The word #space in the for group may be used to define the separator space. Without a separator, all characters are separated (*)
stop	[do]	no	stop execution of the paragraph, stop may have an optional dolist entry:

			program = stop whole program paragraph= stop paragraph, even if in a loop (perRow/combine)
time	do	no	gives date and/or time dependent on the do parameters supplied, see below
unique	do	no	return a list with the unique items
writeLines	for, do	no	writes each parameter of the do group as a line to the file where the file name is given as the for parameter. returns 1 if successful, 0 otherwise

any* = any sequence of words not containing a verb
expression* = arithmetic, logical or comparison expression
(*) note that this may lead to list elements containing a space

11.3 File Operations

these are given as for parameters for the file verb. So far the following operations are implemented:

```
delete = delete the file, if it exists
exists = return 1 if file exists, else 0
create = creates new file, return 1 if successful, else
0
deletedir = delete the directory, if it exists
existsdir = return 1 if directory exists, else 0
createdir = creates new directory, return 1 if
successful, else 0
listdir = returns file list of directory, (use ‘/’ or
’./’ to refer to the current directory)
```

You may specify more than one for parameter and more than one do parameter. The verb proceeds as follows: Each operation is applied to the corresponding file (first op for the first file, second op for the second file ..). If one of the list exhausts while the other still has elements, this list starts over again from the beginning. Thus the total number of executions will be equal to the least common multiple of both list lengths.

11.4 Time Operations

The verb ‘time’ understands the following parameters:

Parameter	Meaning
now	current time
today	current date
+	add the last two dates or times
-	subtract the last two dates or times (x1 x2 -) = x1 - x2
(format definition)	see below, sets the format for all following occurrences of time parameters that are not format defs themselves. No output.
date or time	specified as as string with time or date specific separator(s)

A format definition is a character sequence that contains a sequence of the following shortcuts. The time (clock) separator is always ‘:’.

Shortcut	Meaning
D	day (01..31 or 0..31)
d	day (julian days in time format only)
h	hour (00..24 or 0..24)
m	minute (00..59 or 0..59)
M	month (01..12 or 1..12)
s	seconds (00..59 or 0..59)
Y	year (2 digits)
C	year (4 digits)
w	weekday (1..7) sunday = 7
/	separator
.	separator
b	a space (blank)

z	include leading zeros

11.5 Verb Modifiers

A verb may be augmented by appending certain modifiers to it. Appending is done with a hyphen separating the verb (stem) from the modifier. Note that the modification doesn't necessarily modify the verb action itself but may affect any part of the sentence. We thus can distinguish different kinds of modifiers, which modify the sentence in the following ways:

<input parameters> (a) -> verb action (b) -> <output list> (c)

a) Modifiers affecting the input of the action. i.e. changing the verb parameters before the verb action is applied

b) Modifiers affecting the verb action itself.

c) Modifiers affecting the output, i.e. changing the output of the verb action.

Input modifiers should be given before output modifiers.

Modi-fier	**Type**	**allowed actions**	**Exten-sion?**	**Explanation**
com-bine	b	all	no	Besides the do list, at least two of the three param lists (for, use, with) should be present. The pattern of the do list describes the combination of list elements to be processed in one go. Elements processed are ignored in the next go, until all elements of (at least) one list have been used. Here we use special place holders: #f = for list element, #u = use list element, #w = with list element. If one of these references is invalid, the whole pattern is ignored.
echo	c	all	no	output the parameter list before the action and the result list after the action
echoIn	c	all	no	output the parameter list before the action

echo-Out	c	all	no	output the result list after the action
float	b/c	plist, pexpr	no	treat input as floating point values
forget	own	all	no	after evaluating all params, the list of saved intermediate results and all params are deleted before the action takes place [1])
int	b/c	plist, pexpr, sort	no	treat input as 64-bit integers
perRow	b	all	no	see description below
>name	./.	all	yes	You may give a sentence a name by using>name as a modifier like so: 5 + 7 eval->result. Names must be unique within a paragraph.

[1]) may be used to reduce the amount of memory used in a tail-recursion scenario

Be aware that using modifiers -perRow and -combine tend to be slow. So if possible use plist instead.

11.6 Modifier perRow

Using modifier perRow requires that a use list is present. You may also access a with list and the result list built so far.

The do parameter list describes the actual data to be processed in one step by the verb. It must be divided into two parts, separated by a vertical bar '|'. The parameters before the bar define general characteristics of the action to be taken while the parameters after the bar describe the data to be processed per step. Each use list row is combined with the second part of the do parameter list to produce a new result with the action. The output is the list of these results.

option	part	Ex-ten-sion?	description

row<n>	one	no	n defines the row length, i.e. the number of elements of the use list processed in one step.
def<x>	one	no	default value x to be used for the use list if a referenced index of the use list doesn't exist
wdef<x>	one	yes	default value x to be used for the with list if a referenced index of the with list doesn't exist
rdef<x>	one	yes	default value x to be used for the result list if a referenced index of the result list doesn't exist
init<x>	one	yes	initialize result list with value x, may be repeated to initialize the result list with more than one value.
#	two	no	the nth use list parameter is substituted for the nth place holder sign (#) in the do parameter list
#index	two	no	absolute index of currently processed use parameter
#<n>	two	no	An integer number after the # indicates use list items relative to the current item. #1 has the same meaning as the first occurrence of #. If one of these references is invalid, the whole pattern is ignored.
#w<n>	two	no	same as #<n> but referencing the corresponding item of the with list
#r<n>	two	no	same as #<n> but referencing the corresponding item of the result list
#all	two	no	shorthand for writing #1 #2 ... #n where n is the row length
#rend	two	yes	last element of the result list built up so far

Note that indexes referred to by #, #<n>, #w<n> or #r<n> are given relative to the start of the current step. Thus if you define row3 then the first step processes list elements 1 .. 3, the second 4..6 etc. If the second step is executed, #1 refers to list element 4, #2 to list element 5 etc.

12. Verb References

We distinguish different kinds of verb references:

a) paragraph references
Paragraph references refer to parts of the code having a paragraph name (by which they can be referenced) and trigger the execution defined by that piece of code.

b) value references
Value references refer to parts of the code having a specific value and do not trigger any execution but just return that value.

12.1 Paragraph References

A paragraph reference is denoted by the name of the paragraph to be referenced followed by a dot (as with ordinary verbs). If a paragraph is referenced that way, it is executed. Thus the reference is syntactically regarded as a verb and as such, can have parameters (written before the reference).

12.2 Value References

Value references are denoted by specific keywords which are not verbs (i.e. do not trigger an action) but refer to a value.

word	**scope**	**Exten-sion?**	**Explanation**
this	paragraph	no	refers to the value of the sentence immediately preceding the current sentence
tha*t	paragraph	no	‘th’ followed by successive a’s and a final t. Refers to the value of the (n+1)th sentence preceding the current sentence where n is the number of a’s
args	chapter	no	refers to the parameter list supplied to the file as command line arguments.
params	paragraph	no	This references the actual do parameter

			list of a paragraph call.
forparams	paragraph	no	This references the actual for parameter list of a paragraph call.
useparams	paragraph	no	This references the actual use parameter list of a paragraph call.
withpar-ams	paragraph	no	This references the actual with parameter list of a paragraph call.
ref	paragraph	yes	this word is used for referencing a named sentence and must always have a modifier, which is the sentence name like so: ref~mysentencename.

You may enclose a reference in ||, i.e. |this|. In this case, the list length of the reference is returned instead of the list itself.

Note that a referenced value is never changed by subsequent references to it, even if these include modifiers (see below). Thus a reference in SOPL is not to be understood as a 'call by reference' but always as a 'call by value'.

12.3 Value Reference Modifiers

The meaning of a value reference may be changed by appending certain modifiers to it. Appending is done with a tilde (~) separating the reference from the modifier.

Modi-fier	**allowed referen-ces**	**Explanation**
first	all	only the first list item of the value reference will be referenced
second	all	only the second list item of the value reference will be referenced
last	all	only the last list item of the value reference will be referenced
tail	all	all but the first list item of the value reference will be referenced

head	all	all but the last list item of the value reference will be referenced

It is allowed to use a sequence of these modifiers on a reference. Evaluation is done from left to right. So far all reference modifiers are considered to be standard.

12.4 Paragraph Reference Modifiers

The verb modifiers defined in Chapter 11.5 can be used to modify a paragraph reference.

13. Using Plist execution

13.1 Sentence Syntax

Plist is an ordinary verb that interprets its parameters in a special fashion. The basic call is:

<list to be processed> for <plist opcode list> plist.

There are three versions of plist:
a) use plist to process a list of strings or mixed mode
b) use plist with modifier int to process a list of 64-bit integers
c) use plist with modifier float to process a list of floating point numbers

List elements are processed in a loop from lowest to highest index according to the opcode list as described below. Index incrementation between different iterations depends on the opcodes as described below.

plist expects the dolist to be a special parameter list known as expression. Syntactically they are constants or references modified by operators. Be aware that

- in SOPL **operators do not trigger any code execution**! This is only done by the verbs which take a list of constants, references and operators as parameters. You should rather think of constants and references as nouns and operators as modifiers.
- In SOPL **all operators are postfix operators**, that is they come after the operand(s) they apply to. This means that there is also no need for parentheses.

13.2 Opcode Syntax and Semantics

Plist operates using a stack for intermediate results and uses postfix notation for all operators. Thus to add 5 and 7 you have to input 5 7 +. All operands involved are deleted from the stack and replaced by the result, if there is one. Plist understands the following opcodes:
(Note that when a and b are given as stack operands, b is the top of the stack while a is just below the top).

Opcode	**Exten-sion**	**Explanation**
`x`	no	where x is a string or number: push x on the stack
`row<n>`	no	Gives the number of indexes processed in one iteration. Defaults to one, if not set. Thus row4 would mean that the first iteration starts at index 1 the next at 5 and then 9 etc. **This opcode must be the first opcode.**
`def<n>`	no	sets the default if var<n> would refer to a non-existing index. **Setting the default also switches off the (implicit) restriction to indexes < max. index.** **This opcode must be the second opcode, if set.**
`var<n>`	no	push the value of index n on the stack
`idx`	no	currently processed index number
`and`	no	do a logical and of the top two operands on the stack
`or`	no	do a logical or of the top two operands on the stack
`not`	no	logical not (0 -> 1 else 0)
`==`	no	do a comparison of the two top operands on the stack and push 1 on the stack if equal, 0 otherwise
`<>`	no	do a comparison of the two top operands on the stack and push 0 on the stack if equal, 1 otherwise
`>=`	no	do a comparison of the two top operands on the stack and push 1 on the stack if greater or equal, 0 otherwise (1)
`<=`	no	do a comparison of the two top operands on the stack and push 1 on the stack if less or equal, 0 otherwise (1)
`>`	no	do a comparison of the two top operands on the stack and push 1 on the stack if greater, 0 otherwise (1)
`<`	no	do a comparison of the two top operands on the stack and push 1 on the stack if less, 0 otherwise (1)
`?`	no	elvis operator: if a,b,c are the top three operands where c is the top stack element, then proceed as follows: if a is true (<>0) then the result is b otherwise c.
`+`	no	do an addition (a+ b)
`-`	no	do a subtraction (a - b) of the top two operands on the stack.

`*`	no	Multiply the top two operands of the stack
`/`	no	divide the top two operands of the stack
`%`	no	calculate the modulus a mod b
`&`	no	concatenate the top two operands of the stack (2)
`div`	yes	calculate integer divide and modulus a/b, a mod b and put both on the stack where a mod b is on top. (3)
`max`	no	calculate max(a,b)
`min`	no	calculate min(a,b)
`pwr`	no	a to the power of b
`abs`	no	calculate the abs value of the top operand of the stack
`len`	no	length of the operator in bytes (size) (2)
`out`	no	output the top of the stack and pop it
`drop`	no	pop the top of the stack
`skpz<n>`	no	if the top of the stack is 0, then skip the next n opcodes
`end`	no	stop opcode execution for this round
`uget`	yes	get the top operand from the stack, interpret it as an index to the uselist get the corresponding value and push it on the stack
`wget`	yes	get the top operand from the stack, interpret it as an index to the withlist get the corresponding value and push it on the stack
`isint`	no	get the top operand from the stack, if it is an integer, push “1” on the stack, else “0” (2)
`isnum`	no	get the top operand from the stack, if it is a number, push “1” on the stack, else “0” (2)
`upper`	no	convert string to upper (2)
`lower`	no	convert string to lower (2)

(1) in the string version a check will be made whether both operands are (long) integer, if so, numerical comparison will be used.
(2) only available in string mode
(3) not implemented for the string version

Note: if one of the operators +,*,and,or,min, max has only one operand, no error is thrown and the operator is simply ignored

13.3 Pexpr

The verb pexpr evaluates an expression in postfix notation. It is a stripped down version of plist with the following restrictions/changes:

1. The only parameter list it takes is the do list. All other parameter lists are ignored.
2. The expression given as the do parameter list is only evaluated once.
3. The following opcodes cannot be used with pexpr:
   row<n>, def<n>, var<n>, idx, out, uget, wget.
4. The stack content after executing all operations is the result.

13.4 Further enhancements

The floating point version of plist/pexpr additionally understands the following opcodes:

```
ceil
floor
round
sin
cos
tan
sinh
cosh
tanh
asin
acos
atan
exp
log
log10
sqrt
```

14. Conditional execution

14.1 conditional sequences

Usually conditional execution in a programming language is implemented by some variety of an if/elseif/else statement. In SOPL, however, this feature is implemented by bullet lists (which we call conditional sequences) in the following fashion:

```
a) condition
sentence
...
sentence
b) condition
sentence
sentence
c) condition
sentence
c.1) condition
  sentence
  sentence
c.2) condition
  sentence
  sentence
d) condition
  sentence
  sentence
end)
```

which is to be interpreted as follows:
if the condition in the line with bullet point a) is true, then the following statements are executed until the next line with a bullet point of the same level is encountered (here: b)). Otherwise these statements are skipped and the next bullet point condition of the same level is evaluated.
This process continues until a true condition is found or the bullet list of this level ends. The end is signified by a separate bullet point starting with the word 'end'.

Condition c) contains to conditions c.1) and c.2) on a higher level. This is interpreted as follows: if the c) condition is true, then the first statement after c) is executed. Thereafter, condition c.1) is evaluated and if it is true, it is executed, otherwise, condition c.2) is evaluated and executed if true. The c.2) branch ends at the next bullet point on a lower level (in the case d)).

A paragraph end automatically ends all conditional sequences. That is, conditional sequences cannot span different paragraphs. You nevertheless have to specify an end bullet point to define the end of the condition.

A bullet point comparable to an else branch can be accomplished by a condition that is always true, namely setting condition = 1.

End labels may be defined for all levels, (i.e. end.1.1 defines the end of a 3rd level condition) but only the final end label is mandatory.

14.2 conditional sequence bullet points

A bullet point always ends in a closing parentheses and may contain upper and lower case letters, digits, dots and underscores.

The condition following the bullet point must be written on the same line.

14.3 conditional sequence numbering

Please note, that the numbering of a conditional sequence need not be 'in order'. Also, numbering need not be consistent in the sense that all bullet points follow the same numbering scheme (that is, alphabetic, numeric, roman etc...) The following numbered list is therefore (bad style but) valid:

```
a)
...
f)
...
c)
...
9)
...
III)
```

A dot, however, always introduces a new, higher level of conditional sequencing. Thus the following list is not valid:

a.)
...
b.)
...
c.)

because the sequencing doesn’t start with the lowest level.

14.4 conditional sequence final result

The result of the end) bullet point is the result of the last executed sentence of the condition list.
Note that this will only work as expected if else branches are supplied for all levels. Otherwise the result may be the result of the (last) unsuccessful condition, namely 0.

144.5 conditional sequence evaluation

For the evaluation of conditions the same rules apply as for a pexpr dolist. Usually conditions are evaluated in ‘mixed mode’, allowing you to mix string comparisons with numerical comparisons like:

```
a b > 2 1 > or
```

You may, however, force the given values to be interpreted as integer or float by prefixing the conditional sequence (directly after the bullet point, but separated by a space) with int or float resp. The setting then applies to the whole conditional sequence.

If the condition evaluates to zero, it is considered false, otherwise true.

15. Glossary

char	abbreviation for character (i.e. letter or symbol)
CR	carriage return character
LF	line feed character
ref	reference
reference	see chapter 12
verb	a word that triggers the execution of an action. In SOPL, all predefined verbs are in lower case letters. All arguments (or values) to the verb precede the verb.
white space char	one of the following symbols: space, tab, carriage return, line feed, end of page
word	any sequence of characters, terminated by at least one separator character