BESYQIT

1. Motivation

After implementing SOPL and trying it out for more than half a year it became clear to me that the language wasn't close enough to natural languages yet: Modifiers are not part of the word they are affixed to but remain somewhat separate since they are connected by dashes or tilde.

I furthermore have to admit that English is not a beautiful language (neither in writing nor in pronunciation) so in order to create a beautiful language I had to abandon it and implement a totally different set of verbs/references/modifiers. Instead of trying to use a beautiful language that already exists I decided to invent a new language which I call 'Juubes'. The programming language that was derived from it is called 'Besyqit'. Though vocabulary and grammar are different from SOPL, the basic capabilities are quite similar.
For the purpose of programming the language need not be complete, however, to better motivate the words chosen I supply an introduction to Juubes (see Juubes – Introduction.pdf) and give a basic vocabulary file (juubes-fiiar.csv).

As a result it should be possible to read a program aloud in that language. Note that using a natural language may introduce ambiguities since the same word may be used with different meanings in Juubes. However, Besyqit itself is freed from these ambiguities by defining exact meanings to the words used in a specific context.

I also have to apologize for using English to write a Besyqit manual. An introduction to the Besyqit programming language should rightfully be written in Juubes itself. However, since this language isn't widespread as of now, I hope you forgive me for using an inappropriate natural language.

As with probably all programming languages the grammar and word usage is a little bit different from the 'natural' language itself.

Though the original motivation for inventing Juubes was just to supply words for the programming language Besyqit, it should also be thought of as a spoken language. Since in the standard variety of Besyqit error messages may be given in Juubes, you might want to acquire a basic knowledge of Juubes (or look up the translations in the source code itself).

To avoid confusion: please note that Besyqit uses base30 for numbers (base30.30 cf. Juubes introduction).

2. Basic Structure of Besyqit

A Besyqit program (a file with extension .bsqt or .bsqy) 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.

1. 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 a conjugated form of the verb ‘dii’ (usually ‘diigana’).
2. The header concludes by a call of a paragraph (defined in the body below).
3. The header must be terminated by at least one blank line.
4. The body begins after the blank line that terminates the header and ends at the end of the file.
5. The body is made up of paragraphs, consisting of a paragraph name terminated by a ‘:’, a sequence of sentences and terminated by (at least) one blank line.
6. Paragraph names may only consist of lowercase letters and hyphens. Hyphens cannot appear at the beginning or end of a name and must not follow another hyphen.
7. Sentences are sequences of words terminated by a dot.
8. The last word of a sentence is always a verb. A verb cannot occur in other positions in a sentence and always has a conjugated form (exception: paragraph names are used to define new verbs and in this case are not conjugated, so the paragraph name is a verb stem.)

In the following chapters the term ‘verb’ always refers to a fully conjugated verb (verb stem – optional verb modifiers – verb ending).
The syllable structure of each newly defined paragraph must comply to the structures allowed by Juubes otherwise the paragraph will be rejected, see chapter 6.7.

For programming examples refer to the samples and feature directories.

3. Program Call

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

BESYQIT.exe <options> <besyqit file>

Option	Explanation
-a	set logging level to extended warnings
-dxyz	set logging level to debug and specify debug types, see below
-e	set logging level to error
-f	set logging level to fatal error
-i	set logging level to info
-l	do a lexical analysis of the file and output its result
-L	write all (regular) output to the log file
-n	do not write log output to the console (only to log file)
-p	do a parse of the file and output the result
-prat	set ‘weak‘ option, i.e. - switch off compliance tests - log messages in English - allow use of verb ‘fiiar‘
-r	just read the Besyqit file and echo it back
-t	set logging level to total (log numeric calculations)
-x	execute file
-w	set logging level to warnings

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 verb bai
c	write debug output for paragraph context
i	write debug output for Item (during parse)
j	write debug output for jump calculation
q	write debug output for Juubes conversions
r	write debug output for references
s	write debug output for sentence execution
t	write timing output for loops (~(g)ana, ~(g)ain, bai) 1)
v	write debug output for conditions
x	write debug output for indexes
y	write debug output for setting stop level

1) note that this doesn't work for nested loops

4. Paragraph Structure

A paragraph

- must begin with a word followed by the symbol ':' and a line break.
- It is not allowed to give a paragraph more than one name.
- Paragraph names must be unique within the chapter
- Paragraphs define verb stems. They must be called like all other verbs explicitly by the conjugated 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

5. 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).

6. 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.

6.1. 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

6.2. 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.

6.3. Words

Words cannot contain separator characters and must be separated by separator characters.

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 written number, see chapter 6.4
- 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

    Though arabic numerals are still accepted as input to Besyqit, numbers should be written out using the Besyqit names for digits (see below)
- an operator. Valid operators are defined in chapters 7.1 and 7.3
- 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: \g (tab) \w (newline) \b (double quote) and \\ (backslash). Note, that these escaped characters only appear in console output.

6.4. Numbers in Besyqit

In Besyqit the preferred way of using numbers is to write them as text. Note that Besyqit uses a base30 system.

word	meaning
bu	zero
te	one
da	two
sol	three
swe	four
phoo	five
yaak	six
serif	seven
joor	eight
juban	nine
neel	ten
gitaq	eleven
hubu	twelve
hute	thirteen
huda	fourteen
husol	fifteen
huswe	sixteen
huphoo	seventeen
huyaak	eighteen
huserif	nineteen
hujoor	twenty
hujuban	twentyone
huneel	twentytwo
hugitaq	twentythree
fybu	twentyfour
fyte	twentyfive
fyda	twentysix
fysol	twentyseven
fyswe	twentyeight

fyphoo	twentynine
bute	thirty
sak	(indicate repeat)

It is, however, avoided to repeat the same digit more than once, instead the special marker ‘sak’ is used followed by a digit that states the number of repeats, thus:
tebusaksol = 1000base30 = 27000

6.5. Names

a name may be any of the following
- a verb, see chapter 9
- a paragraph name, see chapter 4 or a reference, see chapter 10
- a part of a grouping sequence, see chapter 8

6.6. Place Holders

Place holder	**Verb stems /Modifiers**	**Explanation**
#	~(g)e,(g)ete	the current list entry
#<n>	~(g)e,(g)ete	the list entry <n> places after the current entry, where <n> is an integer
#sip	~(g)e,(g)ete	the index of the current entry
#fu	kum, ria	a space character
#x,#e,#m	~(g)ain	the next entry in the appropriate list
#chan	~(g)e,(g)ete	shorthand for all indexes of a row

6.7. Compliance

New elements invented by the user must comply to the rules that govern the syllable/word structure of Juubes. This means that paragraph names and sentence names (see chapter 9.3) cannot be chosen arbitrarily. Note that for sentence names, the construction bes + name must also comply. If a name

contains digits, these will be translated one to one to their corresponding Juubes values, thus a123 will be internally translated to ‘atedasol’ not ‘aswesol’. However, other characters are not translated, thus b+123 converts to ‘b+tedasol’ (not ‘bparaitedasol’) and is acceptable, while b123 is not.
For a description of the syllable/word structure rules see chapter 2 of the Juubes introduction.
Checking compliance may be turned of by setting the option -prat (see above).

7. Expressions

Some verbs allow special parameter lists known as expressions. Syntactically they are constants or references linked together or modified by operators. Note than in Besyqit **operators do not trigger any code execution**! This is only done by 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 conjunctions (if they take two arguments) or adjectives (if they take one argument).
Besyqit uses RPN (reverse polish notation) for expressions so the operator follows the operand(s) it refers to.
Three types of operators are implemented:

7.1. Comparison Conjunctions

a comparison is sequence of three words where the first and second word denote a number, string or value reference and the third word is a comparison conjunction, i.e. one of:

```
bass (<)
basycho (>)
seq (= equal)
semu (= not equal)
bosamysoor (<=)
bosamchosoor (>=)
```

To be able to combine comparisons we also introduce:

7.2. Logical Conjunctions/Adjectives

logical conjunctions/adjectives are one of:

```
nea (meaning and)
ren (meaning or)
mu (meaning not)
```

following the operand(s) they refer to.

7.3. Numeric Conjunctions/Adjectives

an expression is a sequence of operands and operators where operands may be numbers, strings or value references and the arithmetic operators:

```
parai (+)
shax (-)
sotai (*)
nymon (/)
mod (%, modulus)
sigoon (&, concat)
ffox (div)
furyy (^, power)
thii (maximum of two elements)
kra (minimum of two elements)
? (elvis operator)
```

with their usual meanings

7.4. Other Numeric Adjectives

```
juu (=absolute value)
ci (=ceiling function)
xo (=floor function)
mul (=rounding function)
sin
cos
tan
sinh
cosh
tanh
asin
acos
atan
exp
log (=natural logarithm)
logneel (= logarithm to base 10)
hor (=square root)
```

with their usual meanings, if not denoted otherwise.

7.5. Other Adjectives

```
vae (=length, size)
haar (=upper case, ‘foreign, strange’)
```

```
lyym (=lower case, ‘beautiful’)
thekah (=is integer)
feewah (=is numeric)
```

8. Parameter Subdivisions

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

They are only used to separate 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.

sequence	**Explanation**
<pl1> janet <pl2> jan	subdivides the parameter list (<pln>) into a janet list and a jan list. The meaning of the lists depends on the verb and the modifier used.
<pl1> janxi <pl2> jan	subdivides the parameter list (<pln>) into a janxi list and a jan list. The meaning of the lists depends on the verb and the modifier used.
<pl1> jannyr <pl2> jan	subdivides the parameter list (<pln>) into a jannyr list and a jan list. The meaning of the lists depends on the verb used.
<pl1> jannyr <pl2> janet <pl3> jan	subdivides the parameter list (<pln>) into a jannyr list, a janet list and a jan list. The meaning of the lists depends on the verb and the modifier used.

Note that these words 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 jan y janet z jannyr a janxi <verb>.

(Note that Besyqit here deviates from Juubes: in Juubes it isn’t possible to suffix the particle ‘jan’ with a case marker.)

9. Verbs

All verbs must be terminated by a dot '.'. It must immediately follow the verb. The following lists give the explanation for all verb stems 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.

9.1. Header Verb Stems

Verb	allowed parameters	Explanation
dii	jan	'include': a list of Besyqit files which may be referenced by this file

9.2. Paragraph Verb Stems

Verb	in English	Call/Explanation
bai	process list	*et* **janet** *nyr* **jannyr bai**+*vmm*. see own chapter
beqo	find	*nyr* **jannyr** *et* **janet beqo**+*vmm*. for each value of the jannyr parameters find the index in the janet parameters (if present) and place it in the result list. **Note that indexing is one-based. The for parameter list should be unique.**
cop	set	*nyr* **jannyr** et **janet cop**+*vmm*. the jannyr parameter list must contain pairs of indexes and values. The corresponding lists elements (by index) in the janet parameter list are then replaced by the given values. The changed list is returned. **Note that indexing is one-based**
dum	input	*et* **janet dum**+*vmm*.

		write the parameters as a prompt followed by ?, read a line from the console as input and return its value
fiiar	convert	*et* **janet fiiar+***vmm*. convert integers and fractional numbers in the janet list to the decimal system, other elements in the list are left unchanged. ***This verb may only be used if the switch -prat is set.***
faon	frequencies	*nyr* **jannyr** *et* **janet faon+***vmm*. the janet list must have the form of a frequencies list = pairs of (keys,occurences). The jannyr list parameter determines the action taken: kum = make list unique with respect to keys thii = return only most frequent pairs, supposes that the list was unified. kra = return only least frequent pairs, supposes that the list was unified.
haan	print	*xi* **janxi haan+***vmm*. the parameters are printed to the console in the sequence in which they are listed. Enclosing double quotes are omitted. The verb recognizes the following escape sequences: \w (wak=back) = newline \g (gaan = move right) = tab \b (bes=speak) = double quote
haanav	output	*xi* **janxi haanav+***vmm*. the parameters are printed “as is” to the console in the sequence in which they are listed
joq	write lines	*xi* **janxi** *et* **janet joq+***vmm*. writes each parameter of the janet group as a line to the file where the file name is given as the janxi parameter. returns 1 if successfull, 0 otherwise
hek	condition	*xi* **janxi hek+***vmm*. only used internally
khi	sort	*nyr* **jannyr** *xi* **janxi khi+***vmm*.

		sort the list of parameters meaning of jannyr parameters, if present: roc = sort descending tut= sort by rows, expects 2 integers following: 1.) index to sort 2.) number of cols per row
khil	process expression	*et* **janet khil**+*vmm*. evaluates an expression using postfix notation. This is a stripped down version of bai, see there.
kum	join	*nyr* **jannyr** *et* **janet kum**+*vmm*. concatenate all Parameters of the janet list by the separator defined by the jannyr list into one. If there is no jannyr list then concatenate without a separator. The word #fu in the jannyr list may be used to define the separator space
kumoryy	unique	*et* **janet kumoryy**+*vmm*. return a list with the unique items
mu	no operation	*xi* **janxi nop**+*vmm*. do nothing. This means that the result list content also isn't changed. Should only be used internally
mud	read lines	*nyr* **jannyr** *xi* **janxi mud**+*vmm*. reads the file and produces as a result a list of lines. You may additionally specify some characteristics with a jannyr list: wayuq moqa = (keepEmpty) an empty line is output as the list element "<wayuq>" wak moqa = (keepEol) every line is followed by the additional list element "<wak>" sec moqa = (keepEof) at the end of the file, the string "<sec>" is added as a list element
nna	set return value	*xi* **janxi nna**+*vmm*. 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.
phe	delete	*nyr* **jannyr** *et* **janet phe**+*vmm*.

		delete the indexes provided in the jannyr parameters from the janet parameters and place the result into the result list **Note that indexing is one-based**
ren	mask	*nyr* **jannyr** *et* **janet ren**+*vmm*. mask all Parameters of the janet group by the corresponding number defined by the jannyr group and return the new list in the following way: the jannyr parameter number gives the number of occurrences of the janxi parameter in the result. If the jannyr parameter list is shorter than the janxi parameter list, it is repeated.
ria	split	*nyr* **jannyr** *et* **janet ria**+*vmm*. split all Parameters of the janet group by the separator defined by the jannyr group and return the new list. If the jannyr parameter contains more than one character, only the first is used. The word #fu in the jannyr group may be used to define the separator space. Without a separator, all characters are separated (*)
roc	reverse	*et* **janet roc**+*vmm*. reverse the list of janet parameters
ryy	expand	*et* **janet ryy**+*vmm*. expand any word of the janet list into a list of its characters. Note that spaces are ignored
sec	stop	*[et* **janet]** **sec**+*vmm*. stop execution of the paragraph, stop may have an optional janet entry: qit = stop whole program gejuc = stop paragraph, even if in a loop as in (g)ain, (g)e
sil	insert	*nyr* **jannyr** *xi* **janxi sil**+*vmm*. the first parameter of the jannyr list is the index after which all subsequent jannyr list parameters are inserted into the janxi parameter list. To insert at the beginning use 0 as the

		index. **Note that indexing is one-based**
silav	minus	*et* **janet** *nyr* **jannyr silav+***vmm***.** determine the set difference between the janet list and the jannyr list (et - nyr)
sip	get	*nyr* **jannyr** *et* **janet sip+***vmm***.** select the indexes provided in the jannyr parameters from the janet parameters and place them into the result list **Note that indexing is one-based. The jannyr parameter list may contain duplicate entries.**
soon	file	*nyr* **jannyr** *et* **janet soon+***vmm***.** the jannyr parameters are the operations that are executed for the files given as the janet parameters, see below
suun	between	*nyr* **jannyr** *xi* **janxi** *et* **janet suun+***vmm***.** insert all parameters of the janxi group between two adjacent parameters of the janet group. if a jannyr group is present, its first element gives the distance between two adjacent inserts. Thus '2 jannyr' inserts after the second, forth,... do parameter.
vae	range: a list of 1 - 3 (integer) pairs	*xi* **janxi vae+***vmm***.** 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)
voos	time	*xi* **janxi voos+***vmm***.** gives date and/or time dependent on the janxi parameters supplied, see below
wah	identity	*xi* **janxi wah+***vmm***.**

		place all janxi parameters into the result list

words in italics do not stand for themselves but for a list of a certain type:
et = janetlist, *xi* = janxilist, *nyr* = jannyrlist, *jan* = janlist
vmm = verb modifiers and marker
expression = arithmetic, logical or comparison expression
(*) note that this may lead to list elements containing a space

9.3. Verb Modifiers and Endings

A verb always requires an ending attached to it. It may also be augmented by appending certain modifiers to it. The sequence for constructing a whole verb is:
<verb stem> <optional modifiers> <ending>

Note that the modification doesn't necessarily modify the verb action itself but may affect any part of the sentence.

Modifier	**allowed actions**	**Explanation**
bool	*	output the parameter list before the action
pri	*	output the result list after the action
fee	khil, bai	treat input as floating point values
khoes	*	after evaluating all params, the list of saved intermediate results and all params are deleted before the action takes place [1])
thek	khil, bai, khi	treat input as 64-bit integers
bes+name	*	You may give a sentence a name by using bes+name as a modifier like so : 5 + 7 khilbesresultana. Names must be unique within a paragraph (check for uniqueness is not yet implemented). If this modifier is present it must be the last in the modifier sequence. **Note**: the name must not end with a 'g'. The construction must comply to Juubes syllable/word

		structure, unless you set the switch -prat (see above)

* = all actions allowed

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

Ending	allowed actions	Explanation
(g)ana	*	'normal' ending. Verb is executed as defined in this chapter
(g)ain	*	Besides the janet list, at least two of the three param lists (jan, janxi, jannyr) should be present. The pattern of the janet 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: #j = jan list element, #n = jannyr list element, #x = janxi list element. If one of these references is invalid, the whole pattern is ignored.
(g)ainab	*	same as (g)ain but the pattern of the janxi list describes the combination of list elements. place holders are #j = jan list element, #n = jannyr list element, #e = janet list element.
(g)e	*	see description below
(g)ete	*	see description below

Be aware that using endings (g)ain or (g)e tend to be slow. So if possible use bai instead.

Note:
To avoid confusion a user defined paragraph must not end with a 'g'.

9.4. Endings (g)e and (g)ete

Using endings (g)e ot (g)ete requires that a jan list is present.

If the ending is (g)e then the janet parameter list describes the actual data to be processed in one step by the verb. If the ending is (g)ete then the janxi parameter list describes the actual data to be processed in one step by the verb.

In both cases the janet/janxi list 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 jan list row is combined with the second part of the janet/janxi parameter list to produce a new result with the action. The output is the list of these results.

option	part	description
tut<n>	one	"row": n defines the row length, i.e. the number of elements of the jan list processed in one step.
jantyy<x>	one	default value x to be used for the jan list if a referenced index of the jan list doesn't exist
nytyy<x>	one	default value x to be used for the jannyr list if a referenced index of the jannyr list doesn't exist
xityy<x>	one	default value x to be used for the janxi list if a referenced index of the janxi list doesn't exist
etyy<x>	one	default value x to be used for the janet list if a referenced index of the janxi list doesn't exist
metyy<x>	one	(meer=result) default value x to be used for the result list if a referenced index of the result list doesn't exist
tas<x>	one	initializes the result list with value <x>. May be repeated to place more than one init value in the result list
#	two	the nth jan list parameter is substituted for the nth place holder sign (#) in the jan parameter list
#sip	two	absolute index of currently processed jan parameter
#<n>	two	An integer number after the # indicates jan 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.
#n<n>	two	same as #<n> but referencing the corresponding item of the jannyr list
#x<n>	two	same as #<n> but referencing the corresponding item of the janxi list
#e<n>	two	same as #<n> but referencing the corresponding item of the janet list
#m<n>	two	same as #<n> but referencing the corresponding item of the result list
#chan	two	shorthand for writing #1 #2 ... #n where n is the row length
#mesec	two	last element of the result list built up so far

9.5. File Operations

these are given as *nyr* parameters for the verb *soon*. So far the following operations are implemented:

```
phe = delete the file, if it exists
wah = (exists) return 1 if file exists, else 0
fawaan = creates new file, return 1 if successful, else 0
rimoo phe = delete the directory, if it exists
rimoo wah = return 1 if directory exists, else 0
rimoo fawaan = creates new directory, return 1 if
successful, else 0
rimoo ria = returns file list of directory, (use ‘/’ or
’./’ to refer to the current directory)
```

You may specify more than one jannyr parameter and more than one janet 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.

9.6. Voos - Time Operations

The concept of ‘measuring’ time and date has only recently been brought to juubesian culture. As a result, date and time are (still) denoted in ‘western’ style using digits and not with juubesian numerals. This may change in the future.

The verb ‘voos’ understands the following parameters:

Parameter	**Meaning**
yue	current time (now)
xab	current date (today)
+	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**
W	(waay=day) day (01..31 or 0..31)
c	(chan = all) day (julian days in time format only)
p	hour (00..24 or 0..24)
m	minute (00..59 or 0..59)
S	(set=moon) month (01..12 or 1..12)
w	(wayuq=empty) seconds (00..59 or 0..59)
j	year (2 digits)

C	year (4 digits)
g	(gejuc= part) weekday (1..7) sunday = 7
/	separator
.	separator
f	a space (blank)
s	(sil= to fill) include leading zeroes

10. References

We distinguish different kinds of references:

a) paragraph references
Paragraph references refer to parts of the code having a 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.

10.1. Paragraph References

A paragraph reference may be denoted by the name of the paragraph to be referenced as part of a verb, where the paragraph name is used as a verb stem.

Paragraph references are action references, that is, 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 and must have an ending.

10.2. 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**	**Explanation**

yii	paragraph	refers to the value of the sentence immediately preceding the current sentence (preceding by 1)
ya	paragraph	refers to the value of the sentence preceding the current sentence by 2
ha	paragraph	refers to the value of the sentence preceding the current sentence by 3
hay	paragraph	refers to the value of the sentence preceding the current sentence by 4
haya	paragraph	refers to the value of the sentence preceding the current sentence by 5
san	chapter	(=args) refers to the parameter list supplied to the file as command line arguments.
ik	paragraph	This references the actual jan parameter list of a paragraph call.
ikyxi	paragraph	This references the actual janxi parameter list of a paragraph call.
iket	paragraph	This references the actual janet parameter list of a paragraph call.
innyr	paragraph	This references the actual jannyr parameter list of a paragraph call.
bes	paragraph	this word is used for referencing a named sentence and must always have a modifier, which is the sentence name like so: besmysentencename.

You may enclose a reference in ||, i.e. |yii|. 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 Besyqit is not to be understood as a ‘call by reference’ but always as a ‘call by value’.

10.3. Reference Modifiers

The meaning of a value reference may be changed by appending certain

modifiers to it. Only value references may be modified.

Modifier	Explanation
ci	(=head) all but the last list item of the value reference will be referenced
wak	only the last list item of the value reference will be referenced
xo	(=tail) all but the first 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.

10.4. Selector Sequence

The meaning of a value reference may also be changed by a following selector sequence. This sequence consists of two words separate from the reference and immediately following it:

sipar <number>
Here sipar is a fixed word while <number> may be any number written in Juubes style (you should not use arabic numerals).

11. Using Bai execution

11.1. Sentence Syntax

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

<list to be processed> janet <bai opcode list> jannyr bai.

There are three versions of bai:
a) use bai to process a list of strings or mixed mode
b) use baithek to process a list of (64-bit) integers
c) use baifee 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. Be aware that ‘mixed mode’ refers to a mixture of string and integer operations, it is not possible to mix string and floating point operations.

11.2. Opcode Syntax and Semantics

et **janet** *nyr* **jannyr bai+***vmm***.**

Bai 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.
Bai 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	**Explanation**
x	where x is a string or number: push x on the stack
tut<n>	(=row) Gives the number of indexes processed in one iteration. Defaults to one, if not set. Thus tut4 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, if set.**
tyy<n>	(=default) sets the default if ik<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.**
sona<x>	(=value) treats the content of x as a value
suo<n>	(=variable) push the value of index n on the stack
sip	(=index) currently processed index number
nea	(=and) do a logical and of the top two operands on the stack
ren	(=or) do a logical or of the top two operands on the stack
mu	(=not) logical not (0 -> 1 else 0)
seq	(==) do a comparison of the two top operands on the stack and push 1 on the stack if equal, 0 otherwise

semu	(<>) do a comparison of the two top operands on the stack and push 0 on the stack if equal, 1 otherwise
bosamchosoor	(>=) do a comparison of the two top operands on the stack and push 1 on the stack if greater or equal, 0 otherwise (1)
bosamysoor	(<=) do a comparison of the two top operands on the stack and push 1 on the stack if less or equal, 0 otherwise (1)
basycho	(>) do a comparison of the two top operands on the stack and push 1 on the stack if greater, 0 otherwise (1)
bass	(<) do a comparison of the two top operands on the stack and push 1 on the stack if less, 0 otherwise (1) Hint: bass basically only means 'to differ'. It is to be understood that the parameters are to be given as an ascending sequence, so the first is lower than the second.
?	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.
parai	(+) do an addition (a+ b)
shax	(-) do a subtraction (a - b) of the top two operands on the stack.
sotai	(*) Multiply the top two operands of the stack
nymon	(/) divide the top two operands of the stack
mod	calculate the modulus a mod b
sigoon	concatenate the top two operands of the stack (2)
ffox	(=div) calculate integer divide and modulus a/b, a mod b and put both on the stack where a mod b is on top. (3)
thii	calculate max(a,b)
kra	calculate min(a,b)
furyy	a to the power of b
juu	(=abs) calculate the abs value of the top operand of the stack
vae	(=len) length of the operator n bytes (size) (2)

pri	(=out) output the top of the stack and pop it
phe	(=drop) pop the top of the stack
jaad<n>	(=skipzero) if the top of the stack is 0, then skip the next n opcodes
sec	(=end) stop opcode execution for this round
xisip	get the top operand from the stack, interpret it as an index to the janxi list get the corresponding value and push it on the stack
ansip	get the top operand from the stack, interpret it as an index to the jan list get the corresponding value and push it on the stack
thekah	(=is integer) get the top operand from the stack, if it is an integer, push “1” on the stack, else “0” (2)
feewah	(=is number) get the top operand from the stack, if it is a number, push “1” on the stack, else “0” (2)
haar	convert string to upper (2)
lyym	convert string to lower (2)

(1) in the string version a check will be made whether both operands are integer (or long), 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 parai, sotai, nea, ren, thii, kra has only one operand, no error is thrown and the operator is simply ignored.

11.3. Khil

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

1. The only parameter list it takes is the janet list. All other parameter lists are ignored.
2. The expression given as the janet parameter list is only evaluated once.
3. The following opcodes cannot be used with khil: tut<n>, tyy<n>, ik<n>, sip, pri, xisip, ansip.
4. The stack content after executing all operations is the result.

11.4. Further Enhancements

There is a floating point version of bai/khil that additionally understands the following opcodes:

```
ci (=round up)
xo (=round down)
mul (=round)
sin
cos
tan
sinh
cosh
tanh
asin
acos
atan
exp
log
logneel (=log10)
hor (=square root)
pi (=3.141592653589793)
```

12. Conditional Execution

12.1. Conditional Sequences

Usually conditional execution in a programming language is implemented by some variety of an if/elseif/else statement. In Besyqit, 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
sec) 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 'sec'.

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.

A bullet point comparable to an else branch can be accomplished by a condition that is always true.

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.

12.2. 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

12.3. 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.

12.4. Conditional Sequence Final Result

The result of the sec) 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 'bu' (zero).

12.5. Executing a Condition

The opcodes allowed in a condition are the same as for the verb khil. You may prefix the condition proper with the words 'thek' and 'fee' to enforce the entries as integer or float. If nothing is specified, all entries are treated as strings.

13. Glossary

char	abbreviation for character (i.e. letter or symbol)
CR	carriage return character
LF	line feed character
paragraph	a user defined piece of code with a name, may be regarded as a function
ref	= reference
reference	see chapter 10
verb	a word that triggers the execution of an action. In BESYQIT, all verbs are in lower case letters. All arguments (or values) to the verb precede the verb. If there are no arguments to the verb but a default argument is defined, then the verb is executed with the default argument. Otherwise, it is executed without an argument. see chapter 9
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, see chapter 6.

14. Translations

We provide a few translations for those words used in Besyqit messages, that are not standard Juubes vocabulary but rather technical terms.

term	**english**	**explanation**
bassydoo	distance	nominalisation of ‘to differ’
benien	word	lit. ‘thing used for language‘
benien (jan) voos	file	lit. ‘river of words’
beqo (jan) loab	user	lit. ‘trying to work’
besav	sentence	lit. ‘small language‘
besav (jan) ik	sentence reference	
booloryy	analyse	lit. ‘strongly inspect’
chanmu	not all, incomplete	
<dibag>	debug	foreign word (english origin)
doo	item	
enojan bes copydoo	jan group marker	
fawaan jan lundoo	definition	
gejuq	part, paragraph	
guas gejuq	verb	lit. ‘strong part’
haanavydoo	output	
hek	condition	
ik	reference	lit. ‘substitute’
ikyy	parameter	lit. ‘strong reference’
jadydoo	modifier	lit. ‘changer’
juryydoo	here: increment	
khil	calculate,execute,process...	
khilydoo sipar te	preprocess	
<kyren>	current	foreign word (english origin)
soon	here: log	
lunakhi	default, standard	
lundatar	directory, syntax tree	

lunykraa	minimum	
meer	result	
mycypan	line	
oorut	vector	
oorut jan thek	index	lit. ‘arrow number’
<parys>	parse	foreign word (english origin)
renydoo	option	lit. ‘alternative’
rimoo	pattern, structure	
rimoodoo	type	nominalisation of ‘categorize’
sago	name	
sago sipar da	label	lit. ‘second name’
secydoo	here: break	
<simbyl>	symbol	
<simbyl> (jan) tutyy	string	sequence of symbols
siparav	level	
sona	value	
suunhi siloo	insert	
taqiv	message, input	
thek	number, version	
thek jan jaadydoo	operator	
tutyy	sequence	
tutyy (jan) taaqi	dictionary, lexicon, lexical	lit. ‘sequence-like book’
wayuq	empty	