14.Type rawstring, string, ustring

The *string* type is used to handle any sorts of string. It provides many different methods to extract a substring, a character or applies any pattern recognition on the top of it.

- The *ustring* type is used to offer a much faster access to very large strings, as the system assumes only one single encoding for the whole string. The “u” stands for “Unicode”. *ustring* is based on the *wstring* implementation in C++.

- The *rawstring* type is quite different. It accepts string but handles them at the byte level. Furthermore, when you create a *rawstring* element, you must provide either its size or an initial string, whose size would be used to bound the variable. The string will not accept characters outside its boundaries, unless you resize it. A *rawstring* does not require specific protection in threads and can be accessed and modified freely. However, you cannot resize a *rawstring* if threads are running in the background. Since, the string is handled at the byte level, the access is very fast, as the system will not try to assess any UTF-8 characters as in the *string* type.

Example:

```
rawstring rd(100);
rd="toto";
println(rd[0],rd[1],rd[2],rd[3],rd[4],rd[5],rd[6]);
//since, the string is managed at the byte level, UTF-8 is not recognized: c l i c h Ã ©
```

14.1.Methods

In the following methods, *rgx* follows the Tamgu regular expression formalism or *TREG* (see the chapter dedicated to these expressions).

1. **base(int b, bool toconvert):** *return the numerical value corresponding to the string numeric content in base b. toconvert is optional. When it is false, the number to be converted is in base 10 and is converted to base b.*

2. **base(vector chrs):** *Set the encoding for each digit in a given base. The default set is 64 characters: 0-9,A-Z,a-z,#,@. Hence, the maximum representation is base 64. You can replace this default character set with your own. If you supply an empty vector, then the system resets to the default set of characters.*

3. **bytes():** *return a ivector of bytes matching the string.*

4. **charposition(int pos):** *convert a byte position into a character position (especially useful in UTF8 strings)*

5. **deaccentuate():** *Remove the accents from accented characters*

6. **doublemetaphone():** *return a svector, which contains the double-metaphone encoding of the string.*
7. **dos():** *convert a string in DOS encoding*
8. **dostoutf8():** *convert a DOS string into UTF8 encoding*
9. **emoji():** *return the textual description (in English) of an emoji.*
10. **evaluate():** *evaluate the meta-characters within a string and return the evaluated string (see below).*
11. **extract(int pos,string from, string up1,string up2...):** *return a svector containing all substrings from the current string, starting at position pos, which are composed of from up to one of the next strings up1, up2,... up1..upn.*
12. **fill(int nb,string char):** *create a string of nb chars.*
13. **find(string sub,int pos):** *Return the position of substring sub starting at position pos*
14. **format(p1,p2,p3):** *Create a new string from the current string in which each '%x' is associated to one of the parameters, 'x' being the position of that parameter in the argument list. 'x' starts at 1.*
15. **geterr():** *Catch the current error output. Printerr and printlnerr will be stored in this string variable.*
16. **getstd():** *Catch the current standard output. Print and println will be stored in this string variable.*
17. **html():** *Return the string into an HTML compatible string or as a vector of strings*
18. **insert(i,s):** *insert the string s at i. If i is -1, then insert s between each character in the input string.*
19. **isalpha():***Test if a string only contains only alphabetical characters*
20. **isconsonant():** *Test if a string only contains consonants*
21. **isdigit():** *Test if a string only contains digits*
22. **isemoji():** *Test if a string only contains emojis*
23. **islower():** *Test if a string only contains lowercase characters*
24. **ispunctuation():** *Test if the string is composed of punctuation signs.*
25. **isupper():***Test if a string only contains uppercase characters*

26. **isutf8():** *Test if a string contains utf8 characters*
27. **isvowel():** *Test if a string only contains only vowels*
28. **last():** *return last character*
29. **latin():** *convert an UTF8 string in LATIN*
30. **left(int nb):** *return the first nb characters of a string*
31. **levenshtein(string s):** *Return the edit distance with s according to Levenshtein algorithm.*
32. **parenthetic():** *Convert a parenthetic expression into a vector (see below)*
33. **parenthetic(string opening, string closing, bool comma, bool separator, bool keepwithdigit, svector rules):** *Convert a recursive expression using opening and closing characters as separators (see below). 'comma' to 'rules' are all optional. See below "tokenization rules" to know how to handle these rules.*
34. **lower():** *Return the string in lower characters*
35. **mid(int pos,int nb):** *return the nb characters starting at position pos of a string*
36. **ngrams(int n):** *return a vector of all ngrams of rank n.*
37. **ord():** *return the code of a string character. Send either the code of the first character or a list of codes, according to the type of the receiving variable.*
38. **parse():** *Parse a string as a piece of code and returns the evaluation in a vector.*
39. **pop():** *remove last character*
40. **pop(i):** *remove character at position i*
41. **read(string file) :** *read a file into the string*
42. **removefirst(int nb):** *remove the first nb characters of a string*
43. **removelast(int nb):** *remove the last nb characters of a string*
44. **replace(sub,str):** *Replace the substrings matching sub with str.* sub *can be a treg.*
45. **reverse():** *reverse the string*
46. **rfind(string sub,int pos):** *Return the position of substring sub backward starting at position pos*
47. **right(int nb):** *return the last nb characters of a string*

48. **scan(rgx, string sep, bool immediatematch, string remaining):** *Return all substrings matching rgx (according to TREG formalism).*

    a. rgx *is a TREG regular expression. According to the recipient variable, it might return a position (int), a sub-string (string), a vector of position (ivector) or a vector of sub-strings (svector).*

    b. sep *is optional. Rgx can be implemented to contain different sub-fields that are separated with 'sep'. In that case, the return value is always a svector.*

    c. immediate*: when 'true' means that the rgx should match starting at the first character. When 'false', the default value, scans the whole string to find the first match. Must be used with sep.*

    d. remaining: *must be a string variable. When provided, it returns the rest of the string after the section that matches. Must be combined with* immediate *and* sep.

49. **size():** *return the length of a string*

50. **slice(int n):** *slice a string into substring of size n.*

51. **split(string splitter):** *split a string along splitter and store the results in a svector (string vector). If splitter=="", then the string is split into a vector of characters. If splitter is not provided, then the string is split along space characters.* splitter *can be a treg. If the splitter is a* preg *then the split is done according to the whole posix regular expression, not as a substring to detect.*

52. **splite(string splitter):** *split a string the same way as split above, but keep the empty strings in the final result. Thus, if "s='+T1++T2++T3", then s.split("+") will return ["T1","T2","T3"], while s.splite("+") will return ["","T1","","T2","","T3"]. If the splitter is a* preg *then the split is done according to the whole posix regular expression, not as a substring to detect.*

53. **multisplit(string sp1, string sp2…):** *split a string along multiple splitter strings.*

54. **stokenize(map keeps):** *Tokenize a string into words and punctuations. Keeps is used to keep together specific strings.*

55. **tags(string o,string c, bool comma, bool separator, bool keepwithdigit, svector rules):** *Parse a string as a parenthetic expression, where the opening and closing strings are provided. 'comma' to 'rules' are all optional. See below "tokenization rules" to know how to handle these rules.*

56. **tokenize(bool comma, bool separator, svector rules):** *Tokenize a string into words and punctuations. If comma is true, then the "," is the decimal separator, otherwise it is the ".". If 'separator' is true, then '.' or ',' can be used as separators as in: "3,000.10". tokenize returns a svector. Each of these parameters is optional.*

When one of these parameters is omitted, then its default value is false. See below "tokenization rules" for how to use this parameter.

57. **trim():** *remove the trailing characters*

58. **trimleft():** *remove the trailing characters on the left*

59. **trimright():** *remove the trailing characters on the right*

60. **upper():** *Return the string in upper characters*

61. **utf8():** *convert a LATIN string into UTF8*

62. **utf8(int part):** *convert a Latin string, encoded into ISO 8859 part part into utf8. For instance, s.utf8(5), means that the string to be converted in UTF-8, is encoded in ISO 8859 part 5 (Cyrillic). See below for a description of each part.*

63. **write(string file):** *write the string content into a file*

14.2.String Handling

There are a number of methods and implementation that are specific to strings.

Korean specific methods (only for string and ustring)

1. **ishangul():** *return true if it is a Hangul character.*

2. **isjamo():** *return true if it is a Hangul jamo.*

3. **jamo(bool combine):** *if 'combine' is false or absent: split a Korean jamo into its main consonant and vowel components, else combine content into a jamo.*

4. **normalizehangul():** *Normalize different UTF8 encoding of Hangul characters*

5. **romanization():** *Romanization of Hangul characters.*

Latin Table

(from https://en.wikipedia.org/wiki/ISO/IEC_8859)

Use the number associated to "Part" in the first column with the utf8 method.

Part 1	Latin-1 Western European	Perhaps the most widely used part of ISO/IEC 8859, covering most Western European languages: Danish (partial),[1] Dutch (partial),[2]English, Faeroese, Finnish (partial),[3] French (partial),[3] German, Icelandic, Irish, Italian, Norwegian, Portuguese, Rhaeto-Romanic,Scottish Gaelic, Spanish, Catalan, and Swedish. Languages from other parts of the world are also covered, including: Eastern EuropeanAlbanian, Southeast Asian Indonesian, as well as the African languages Afrikaans and Swahili. The missing euro sign and capital Ÿ are in the revised version ISO/IEC 8859-15 (see below). The corresponding IANA character set is ISO-8859-1.
Part 2	Latin-2 Central European	Supports those Central and Eastern European languages that use the Latin alphabet, including Bosnian, Polish, Croatian, Czech, Slovak, Slovene, Serbian, and Hungarian. The missing euro sign can be found in version ISO/IEC 8859-16.
Part 3	Latin-3 South European	Turkish, Maltese, and Esperanto. Largely superseded by ISO/IEC 8859-9 for Turkish and Unicode for Esperanto.
Part 4	Latin-4 North European	Estonian, Latvian, Lithuanian, Greenlandic, and Sami.
Part 5	Latin/Cyrillic	Covers mostly Slavic languages that use a Cyrillic alphabet, including Belarusian, Bulgarian, Macedonian, Russian, Serbian, andUkrainian (partial).[4]
Part 6	Latin/Arabic	Covers the most common Arabic language characters. Doesn't support other languages using the Arabic script. Needs to be BiDi andcursive joining processed for display.
Part 7	Latin/Greek	Covers the modern Greek language (monotonic orthography). Can also be used for Ancient Greek written without accents or in monotonic orthography, but lacks the diacritics for polytonic orthography. These were introduced with Unicode.
Part 8	Latin/Hebrew	Covers the modern Hebrew alphabet as used in Israel. In practice two different encodings exist, logical order (needs to be BiDi processed for display) and visual (left-to-right) order (in effect, after bidi processing and line breaking).
Part 9	Latin-5 Turkish	Largely the same as ISO/IEC 8859-1, replacing the rarely used Icelandic letters with Turkish ones.

Part 10	Latin-6 Nordic	a rearrangement of Latin-4. Considered more useful for Nordic languages. Baltic languages use Latin-4 more.
Part 11	Latin/Thai	Contains characters needed for the Thai language. Virtually identical to TIS 620.
Part 12	Devanagari	The work in making a part of 8859 for Devanagari was officially abandoned in 1997. ISCII and Unicode/ISO/IEC 10646 cover Devanagari.
Part 13	Latin-7 Baltic Rim	Added some characters for Baltic languages which were missing from Latin-4 and Latin-6.
Part 14	Latin-8 Celtic	Covers Celtic languages such as Gaelic and the Breton language.
Part 15	Latin-9	A revision of 8859-1 that removes some little-used symbols, replacing them with the euro sign € and the letters Š, š, Ž, ž, Œ, œ, and Ÿ, which completes the coverage of French, Finnish and Estonian.
Part 16	Latin-10 South-Eastern European	Intended for Albanian, Croatian, Hungarian, Italian, Polish, Romanian and Slovene, but also Finnish, French, German and Irish Gaelic(new orthography). The focus lies more on letters than symbols. The currency sign is replaced with the euro sign.

The “Part” number is the index through which encodings are accessed.

Note: The table *part* 17, which is not mentioned here, is an addendum to handle “Windows 1252 Latin 1 (CP1252)” encoding.

Encoding Names

Since, using a number to refer to the right encoding is quite cumbersome, Tamgu provides the following constant value to access these encodings:

• e_latin_we =1 :	*Western European*
• e_latin_ce = 2:	*Central European*
• e_latin_se = 3:	*South European*
• e_latin_ne = 4:	*North European*
• e_cyrillic = 5:	*Cyrillic*
• e_arabic = 6:	*Arabic*
• e_greek = 7:	*Greek*
• e_hebrew = 8:	*Hebrew*
• e_turkish = 9:	*Turkish*
• e_nordic = 10:	*Nordic*
• e_thai = 11:	T*hai*

- e_baltic = 13: *BALTIC RIM*
- e_celtic= 14: *Celtic*
- e_latin_ffe= 15: *Extended (French, Finnish, Estonian)*
- e_latin_see= 16: *South East European*
- e_windows = 17: *Windows encoding*
- e_cp1252 = 17: *Windows encoding (to match the exact name)*

Meta-characters

If you use strings declared between “”, then Tamgu will automatically recognize the following meta-characters:

- \n, \r and \t which are the line feed, the carriage return, and the tabulation respectively.

Function *evaluate*

Tamgu also recognizes another large set of meta-characters, which are automatically translated for you when you use the method “*evaluate*”:

- Decimal code: \ddd, which is then translated into the Unicode character of that code: \048 is for instance the character ‘0’.

- Hexadecimal code: \xhh, which is also translated into the corresponding Unicode character: \x30 is the character ‘0’.

- Unicode code: \uhhhh, which is also translated into the corresponding Unicode character: \u0030 is the character ‘0’.

- &#d(d)(d)(d); which is also translated in the corresponding Unicode character: &#30; is the character ‘0’. This coding occurs in XML and HMTL texts.

- &namecode; for which a long list of equivalence exists (XML and HTML again). For instance: &eaccute; is the character: é.

Conversely, the method “html” returns a string in which non ASCII character are translated into HTML encoding.

Emojis

Tamgu also keeps a track of emojis (*V.5 beta* Unicode 2017), whose list can be gathered with the procedure: *emojis()*, which returns a *treemapls* object, where the key is the emoji Unicode and the value its textual description in English. Furthermore, Tamgu provides two methods *isemoji* and *emoji*, which indicates whether a string is composed of emojis or their description.

Operators

sub in s: test if sub is a substring of s. If *sub* is a TREG and the recipient variable a *ivector* then Tamgu returns both the beginning and the end of the strings that were detected with the regular expression.

for (c in s) {…}: loop among all characters. At each iteration, c contains a character from s.

+: concatenate two strings.

"…": define a string, where meta-characters such as "\n","\t","\r","\"" are interpreted.

'…': define a string, where meta-characters are not interpreted. This string cannot contain the character "'".

Indexes

str[i]: return the ith character of a string

str[i:j]: return the substring between i and j. i and j can be substrings, which the system will use to extract the substring.

str[s..]: return the substring starting at string s.

str[-s..]: return the substring starting at string s. In this case, s is searched from the end of the string.

N.B. When i and j are positive integers, they are treated as absolute positions within the string. However, when the values are negative, they are considered as offsets to be counted from each string extremities. However, if the first element of the interval is a substring and the second one is a positive integer, then this second index will be treated as an offset from the rightmost position of where the substring was found.

You can also modify a character range.

Example:

```
string s="This is a cliché, which contains a 'é'";

s[10:16]        cliché                      //absolute positions

s["cliché":7]   cliché, which               //offset from end of substring

s["cliché":-4]  cliché, which contains a    //offset from end of string

s[-"a":]        a 'é'   //looking for last instance of a

s[-"a":]="#"    This is a cliché, which contains #  //replacing a substring
```

If an index is out of bounds, then an exception is raised unless the flag *erroronkey* has been set to *false.* In that case, Tamgu will return *empty.*

As an integer or a float

If the string contains digits, then it is converted into the equivalent number, otherwise its conversion is 0.

format

A format string contains specific variables, which can be replaced on the fly with some content.

```
string frm="this %1 is a %2 of %1 with %3";

str=frm.format("test",12,14);
println(str); //Result: this test is a 12 of test with 14
```

scan

There are two different versions for *scan*.

The first version takes only one argument and applies the regular expression (*TREG*) across the whole string, extracting every single target that matches the *TREG*. Each element in that case *corresponds to the whole regular expression*.

The second version takes a separator. This version of scan considers the regular expression as extracting different fields separated with a separator.

For instance: scan("%d+,%d+",',') considers expressions in which there are two integers separated with a “,”. This expression will then return two elements for: *12,34,45,56* and only two: *12,34.*

Note also, that this version *does not* return positions for ivector as for the other versions.

Hence: ivector iv = scan("%d+,%d+",',');

Will return: [12,34], the values themselves…

The first method simply splits the string along the regular expression, while the second one interprets the content of that string.

```
//A macro to read complex hexadecimal structures

grammar_macros("X","({%+-})0x%x+(.%x+)(p({%+-})%d+)");
```

```
string s="This: 0x1.0d0e44bfeec9p-3 0x2.09p3 0x3.456aebp-1 in here.";

//We use the macro
string res=s.scan("%X");
println("Res:",res);  //Res: 0x1.0d0e44bfeec9p-3


ivector iv = s.scan("%X");
println("IV",iv); //IV [6,25,26,34]


svector vs=s.scan("%X");
println("VS",vs); //VS ['0x1.0d0e44bfeec9p-3','0x2.09p3',' 0x3.456aebp-1'] //3 elements

 //with a separator... The difference here is that
//the two numbers should be separated with a space character

vs = s.scan("%X %X"," ");
println("VS",vs); //VS ['0x1.0d0e44bfeec9p-3','0x2.09p3'] //2 elements…

string reste;
fvector fv = s.scan("%X, %X",",",false,reste) ;
println("FV",fv, reste); //FV [0.131375,16.2812] 0x3.456aebp-1 in here.
```

treg string or not _treg_ string?

In all the examples that have been shown so far, *scan* takes as input a string, which is then compiled into a *treg*. It is actually possible to provide a *treg* instead of a string as the first parameter of scan. If this *treg* is given as a *r* string, then the *treg* will be compiled at parse time and not at execution time. Thanks to this pre-compiling, there is a slight advantage in using *treg* instead of strings at runtime.

Tokenization Rules

The methods: *parenthetic, tags* and *tokenization* all use an underlying set of tokenization rules, which can be modified through their *rules* parameter.

This underlying set of rules can be loaded and modified to change or enrich the tokenization process, thanks to *_getdefaulttokenizerules.*

```
svector rules=_getdefaulttokenizerules();
```

The rules are applied according to a simple algorithm. First, rules are automatically identified as:

- character rules: the rule starts with a specific character
- entity rules: the rule starts with an entity such as: %a, %d etc…
- metarules: the rule pattern is associated with an id that is used in other rules.

The rules should always be ordered with character rules first and ends with entity rules. The most specific rules should precede the most general ones.

Metarules

A metarule is composed of two parts: c:expression, where c is the metacharacter that is accessed through %c and expression is a single body rule.

for instance, we could have encoded %o as: "o:[≠ ∨ ∧ ÷ × ² ³ ¬]"

IMPORTANT: These rules should be declared with one single operation.

Their body will replace the call to a %c in other rules (see the test on metas in the parse section)

If you use a character that is already a meta-character (such as "a" or "d"), then the meta-character will be replaced with this new description... However, its content might still use the standard declaration:

"1:{%a %d %p}": "%1 is a combination of alphabetical characters, digits and punctuations

Rules

A rule is composed of two parts: *body=action*.

- *action* is either an integer or a #, which can have a specific meaning for the tokenizer. For instance, number descriptions come with a '9' as their action descriptor.
- # means that the extracted string will not be stored for parsing (spaces, cr and comments mainly)

body uses the following instructions:

- x is a character that should be recognized
- #x-y comparison between x and y. x and y should be ascii characters...
- %x is a meta-character with the following possibilities:
  - %. is any character
  - %a is any alphabetical character (including unicode ones such as éè)
  - %C is any uppercase character
  - %c is any lowercase character
  - %d is any digits

- %H is any hangul character
- %n is a non-breaking space
- %o is any operators
- %p is any punctuations
- %r is a carriage return both \n and \r
- %s is a space (32) or a tab (09)
- %S is both a carriage return or a space (%s or %r)
- %? is any character with the possibility of escaping characters with a '\' such as: \r \t \n or \"
- %nn you can create new metarules associated with any characters...

- (..) is a sequence of optional instructions
- [..] is a disjunction of possible characters
- {..} is a disjunction of meta-characters
- x+ means that the instruction can be repeated at least once
- x- means that the character should be recognized but not stored in the parsing string
- %.~.. means that all character will be recognized except for those in the list after the tilde.

IMPORTANT: do not add any spaces as they would be considered as a character to test...

Example

```
svector rules=_getdefaulttokenizerules();
rules.insert(55,"{%a %d}+ : {%a %d}+=0"); // aaa : bbb is now one token
rules.insert(55,"{%a %d}+.{%a %d}+=0");  // aaa.bbb is now one token
rules.insert(38,"->=0"); // -> is one token

string s="this is a test.num -> x : 10 ";

//Without rules
v= s.tokenize(); //['this','is','a','test','.','num','-','>','x',':','10']

//With rules
v= s.tokenize(false,false, regles); //['this','is','a','test.num','->','x : 10']
```

parenthetics() or parenthetics (string opening, string closing)

Tamgu also provides a way to decipher parenthetic expressions such as:

```
( (S (NP-SBJ Investors)
    (VP are
       (VP appealing
          (PP-CLR to
                     (NP-1 the Securities))
          (S-CLR (NP-SBJ *-1)
                    not
                    (VP to
                         (VP limit
                            (NP (NP their access)
                                 (PP to
                                    (NP (NP information)
                                         (PP about
                                            (NP (NP stock purchases)
                                                 (PP by
                                                    (NP "insiders")
)))))))))).))
```

Tamgu provides a method: *parenthetics* which takes as input a structure as the one above and translates it into a *vector.*

```
vector v=s. parenthetics(); //s contains a parenthetic expression as above
```

The second function enables the use of different opening or reading characters.

Example:

Tamgu can analyze the structure below:

```
< <S <NP-SBJ They>

    <VP make

       <NP the argument>

         <PP-LOC in

                 <NP <NP letters>

                    <PP to

                         <NP the agency>> > > > > .>
```

with the following instruction:

```
vector v=s. parenthetics('<','>');
```

tags(string opening, string closing)

tags is similar to the *parenthetics* method except that instead of characters, it takes strings as input*. You should not use this method to parse XML output, use* xmldoc *instead.*

```
string s="OPEN This is OPEN a nice OPEN example CLOSE CLOSE CLOSE";
vector v=s.tags('OPEN','CLOSE');
```

```
Output: v=[['this', 'is', ['a','nice', ['example']]];
```

14.3.Examples

```
//Below are some examples on string manipulations
string s;
string x;
vector v;

//Some basic string manipulations
s="12345678a";
x=s[0];             // value=1
x=s[2:3];                   // value=3
x=s[2:-2];                  //value=34567
x=s[3:];            //value=45678a
x=s[:"56"];                 //value=123456
x=s["2":"a"];               //value=2345678a
s[2]="ef";                  //value=empty

//The 3 last characters
x=s.right(3);               //value=78a

//A split along a space
s="a b c";
v=s.split(" ");             //v=["a","b","c"]


//regex, x is a string, we look for the first match of the regular expression
x=s.scan("%d%d%c");   //value=78a

//We have a pattern, we split our string along that pattern
s=’12a23s45e’;
v=s.scan(r"%d%d%c");             // value=['12a','23s','45e']
x=s.replace(r"%d%ds","X");       //value=12aX45e

//replace also accepts %x variables as in Tamgu regular expressions
x=s.replace(r"%d%1s","%1");      //value=12a2345e

//REGULAR REGULAR EXPRESSIONS: Not available on all platforms
```

```
preg rgx(p'\w+day');
string str="Yooo Wesdenesday Saturday";
vector vrgx=rgx in str;    //['Wesdenesday','Saturday']

string s=rgx in str;          //Wesdenesday
int i=rgx in str;    // position is 5

//We use (…) to isolate specific tokens that will be stored in the
//vector
rgx=p'(\d{1,3}):(\d{1,3}):(\d{1,3}):(\d{1,3})';
str='1:22:33:444';
vrgx=str.split(rgx);          // [1,22,33,444], rgx is a split expression

str='1:22:33:4444';
vrgx=str.split(rgx);          //[] (4444 contains 4 digits)

str="A_bcde";
//Full match required
if (p'[a-zA-Z]_.+' == str)
    println("Yooo");          //Yooo

//this is also equivalent to:
rgx = p'[a-zA-Z]_.+';
if (rgx.match(str))
    println("Yooo bis");

str="ab(Tamgu12,Tamgu14,Tamgu15,Tamgu16)";
vector v=str.extract(0,"Tamgu",",",")"); //Result: ['12', 14',' 15',' 16']

string frm="this %1 is a %2 of %1 with %3";

str=frm.format("tst",12,14);
println(str); //Result: this tst is a 12 of tst with 14
```