Tamgu (탐구) Language

1 Table of Contents

2 Summary

This document describes the Tamgu language.

At its core, Tamgu is an imperative language, enriched with a predicate engine, inspired from Prolog, and functional capabilities inspired from the HASKELL language. The three styles of programming can be freely mixed together.

Tamgu also provides a way to enrich the language with external libraries. Actually some of them are actually described in this document. For instance, Tamgu provides Graphical Capabilities, Sound capabilities and data base management, through an encapsulation of SQLite.

The language is available on most platforms: Microsoft Windows, Mac OS and different flavors of UNIX.

For any questions or any problems related to the language, you can send a mail to:

claude.roux@naverlabs.com

3 Tamgu: an introduction

The Tamgu language borrows many concepts from many other languages, mainly C++ and Python. It is therefore quite straightforward to learn for someone with a basic knowledge of theses languages.

3.1 Some elements of language

A Tamgu program contains variable declarations, function declarations and frames (or classes) declarations. A variable can be declared anywhere at any place, the same applies to functions, to the exception of loops.

Comments

Comments for a line are introduced anywhere with a //.

```
//This is a comments
```

Comments for a bloc of lines are inserted into: /@...@/

```
/@
This is…
a bloc of comments.
@/
```

Function

A function is declared with the keyword *function*, a name and some parameters.

Frame

A frame is declared with the keyword *frame*, followed with a name. A sub-frame is simply declared as a frame within a frame.

Function and frame pre-declarations

The pre-declaration of functions and frames is not necessary in Tamgu, since the interpreter first loops into the code to detect all functions and frames and declares them beforehand.

Hence:

```
//We call call2 from with call1
function call1(int x, int y) {
    call2(x,y);
}

//call2 is declared after call1
function call2(int x,int y) {
    println(x,y);
}
```

is a perfect licit code.

3.2 System Functions

Exit: _exit()

This function is used to exit definitely from a program.

Error on key: _erroronkey(bool)

By default, any attempt to access a value in a map with an unknown key *does not raise an exception*. The function: _*erroronkey(bool),* which should be placed at the very beginning of your code modifies this behavior.

Stack Size: _setstacksize(size)

The stack size is initially set to 1000 functions calls. You can modify this value with this function. However, if your stack size is too large, then your program might crash as it could become larger than the actual stack size of your system.

Number of threads: _setthreadlimit(nb)

The number of actual threads that can run in parallel is initially set to 1000 on Windows and 500 on other systems. You can modify this value to increase the number of threads that can run in parallel.

Valid features: _setvalidfeatures(mapss features)

This method is used to put some constraints on the valid features that can be used both for *synodes* and *dependencies*.

See the *synode, dependency* section for more details.

Number of threads to be joined together: _setjoinedlimit(nb)

By default up to 256 threads can be "joined" together. You can modify this number with this function.

Initial environment variables: _setenv(varname,value)

It is possible to set environment variables at launch time with this function.

Tamgu version: _version()

Returns a string with version information about Tamgu.

Mirror display: _mirrordisplay(bool)

This function is used to set the mirror display from within a GUI. When it is activated "print" displays values both on the GUI output and the command window output.

Memory management: _poolstats()

This function is used to return the current state of different object pools. For efficiency reasons, some objects are managed in pools, which allows for a reusability of recurrent objects (such as strings, integers, floats, vectors or maps). These objects are not deleted but cleared and yielded back for reuse when needed. The result of this function is a *mapsi* object.

3.3 Passing arguments to a Tamgu program

When a Tamgu program is called with a list of arguments, each of these arguments becomes available to the Tamgu program through three specific systems variables: *_args, _current* and *_paths.*

Example:

```
tamgu myprogram c:\test\mytext.txt
```

_args: Argument vector

Tamgu provides a specific variable: "*_args*", which is actually a string vector in which each argument is stored according to its position in the declaration list.

Example (from the call above):

```
file f;
f.openread(_args[0]);
```

_paths,_current: Path management

Tamgu provides a second vector variable: *_paths*, which stores the pathnames of the different *Tamgu* programs, which have been loaded.

```
//Displaying all paths loaded in memory
string n;
for (n in _paths)
  print("Loaded: ",n,"\n");
```

Important

The first element of this vector: *_paths[0]* stores the current directory pathname. *_paths[1]* stores the path of the current program file.

_current

_current is another interesting variable that stores the path of the program file that is currently being run. The path stored in *_current* always finishes with a final separator. Actually, *_current* points to the same path as *_paths[1]*.

_endl: Carriage return

Windows and Unix do not use exactly the same carriage return, as on Windows, a carriage return is usually two characters long: “\r\n”.

_endl returns the proper carriage return according to the platform value.

_sep : Separator in pathnames

Unix-based systems and Windows use different separators in pathnames, between directory names. Unix requires a “/” while Windows requires a “\”.

Tamgu provides a specific variable: *_sep*, which returns the right separator according to your system.

3.4 Console

Tamgu provides a default console, which can be used to load and edit any programs. The console can be used to test small pieces of code or to check the values at the end of an execution.

You can also execute a program in a debug mode, which then displays the content of the stack and of the variables at each step in your program.

To launch the console, run *tamgu* with: *-console.*

4 Command line (Unix platforms)

You can use Tamgu as a command line tool on Linux or Mac OS platforms.

Tamgu offers the following commands:

- -i: this command allows you to pre-declare the variables:
bool a,b,c; int i,j,k; float f,g,h; string s,t,u; map m; vector v; self x,y,z;

- -c: Allows you to execute a piece of tamgu code provided after the order.
  Example: tamgu –c ' println(10+20);'

- -a: intercepts the sendings on stdio and records them in the vector _args. It is possible to provide a piece of code.

- -p: intercepts line-by-line sending on stdio and allows the execution of a piece of code where the pre-declared variable l contains the current stdio line.
  Example: ls -l | tamgu –p ' println(l);'

Tamgu can also be called with a program name and a list of arguments. This list of arguments is then available in _args.

Example: tamgu mycode.tmg a1 a2 a3 a3
_args contains: ["a1", "a2", "a3"]

It is also possible to combine the stdio lines and the call of a program with its arguments. The "-a" option must then be placed after the program name:

Example: ls -1 | tamgu mycode.tmg ***-a*** a1 a2 a3

4.1 Integrated IDE

Tamgu provides an integrated text mode IDE that you can use to create, modify or even debug your programs.

You can call this environment in two different ways.

a) If you run "tamgu" without arguments, you can then type the "edit" command which will put you in edit mode.

b) You can also launch: “tamgu -e myfile.tmg” which will open your file and automatically put you in edit mode. If "myfile.tmg" does not exist, you can create it the first time you save it.

Edit/Run mode

You can use Tamgu either to edit your file or execute them. You can launch the execution of your file from within the editor with: Ctrl-Xr. Once your code has been executed, you will switch to the “run mode” where you can test your variables. You can also switch to the “run mode” with “Ctrl-c”.

To switch back to “edit mode” type: edit.

Help

The IDE offers many commands including the following online help, which can be called either with the command “help” in run mode or with “Ctrl-Xh” in edit mode.

```
Commands:

 - 1. Programs:
   - create filename: create a file space with a specific file name
   - open filename: load a program (use run to execute it)
   - open: reload the program (use run to execute it)
   - save filename: save the buffer content in a file
   - save: save the buffer content with the current filename
   - run filename: load and run a program filename
   - run: execute the buffer content
   - debug: debug mode
      - n!: next line
      - l!: display local variables
      - a!: display all active variables
      - s!: display the stack
      - g!: go to next breakpoint
      - e!: execute up to the end
      - S!: stop the execution
      - name: display the content of a variable (just type its name)
      - h!: display debug help
      - args v1 v2...vn: set a list of values to the argument list
      - filename: display the current filename

 - 2. Command line mode:
   - help: display the help
   - help n: display one of the help sections (from 1 to 5)
   - history: display the command history
   - load filename: load a command history file
   - store filename: store the command history in a file
   - filename: display the current file name
   - filespace: display all the files stored in memory with their
     file space id
   - select idx: select a file space
   - metas: display meta-characters
```

  - **rm:** clear the buffer content
  - **rm b:e:** remove the lines from b to e (b: or :e is also possible)
  - **list:** list the buffer content
  - **list b:e:** display the buffer content from line b to e (b: or :e is also possible)
  - **Ctrl-t:** up in the code listing
  - **Ctrl-g:** down in the code listing
  - **Ctrl-f:** force the current line to be appended at the end of the code
  - **?method:** return a description for 'method'
  - **?o.method:** return a description for 'method' for variable 'o'

- **3. edit (idx):** edit mode. You can optionally select also a file space
  - **Ctrl-b:** toggle breakpoint
  - **Ctrl-k:** delete from cursor up to the end of the line
  - **Ctrl-d:** delete a full line
  - **Ctrl-u:** undo last modification
  - **Ctrl-r:** redo last modification
  - **Ctrl-f:** find a string
  - **Ctrl-n:** find next
  - **Ctrl-g:** move to a specific line, '$' is the end of the code
  - **Ctrl-l:** toggle between top and bottom of the screen
  - **Ctrl-t:** re-indent the code
  - **Ctrl-h:** local help
  - **Ctrl-w:** write file to disk
  - **Ctrl-c:** exit the editor

  - **Ctrl-x:** ***Combined Commands***
    - *C:* count a pattern
    - *H:* convert HTML entities to Unicode characters
    - *D:* delete a bloc of lines
    - *c:* copy a bloc of lines
    - *x:* cut a bloc of lines
    - *v:* paste a bloc of lines
    - *d:* run in debug mode
    - *r:* run the code
    - *w:* write and quit
    - *l:* load a file
    - *m:* display meta-characters
    - *h:* full help
    - *q:* quit

- **4. Environment:**
  - **cls:** clear screen
  - **color:** display terminal colors
  - **color att fg bg:** display a color out of attribute, foreground, background
  - **colors:** display the colors for code coloring
  - **colors name att fg bg:** modify the color for one of the following denominations = *strings, methods, keywords, functions, comments*

- **5. System:**
  - **!unix:** what follows the *'!'* will be executed as a Unix command (ex: *!ls*)
  - **clear (idx):** clear the current environment or a specifc file space
  - **reinit:** clear the buffer content and initialize predeclared variables
  - **Ctrl-d:** end the session and exit tamgu

- **exit:** end the session and exit Tamgu

- **6. Regular expressions for '*find*'**

%d stands for any digit
%x stands for a hexadecimal digit (abcdef0123456789ABCDEF)
%p stands for any punctuation
%c stands for any lowercase letter
%C stands for any uppercase letter
%a stands for any letter
? stands for any character
%? stand for the character “?” itself
%% stand for the character “%” itself
%s stand for any space character include the non-breaking space
%r stand a carriage return
%n stand for a non-breaking space
~ negation

\x	escape character
\ddd	character code across 3 digits exactly
\xFFFF	character code across 4 hexas exactly
{a-z}	between a and z included
[a-z]	sequence of characters

^ the expression should start at the beginning of the string
$ the expression should match up to the end of the string

Examples:

dog%c	**matches dogs or dogg**
m%d	**matches m0, m1,…,m9**
{%dab}	**matches 1, a, 2, b**
{%dab}+	**matches 1111a, a22a90ab**

5 Basic Types

Tamgu requires all items to be declared with a specific type. Types are either predefined or user-defined as a frame.

5.1 Predefined types

Tamgu proposes some very basic types:

Basic Objects:

self, string, int, decimal, float, long, bit, short, fraction, bool, date, time, call

Containers:

vector, map, imatrix, fmatrix, file, iterator

function

A function is declared anywhere in the code, using the keyword *function*.

frame

A *frame* is a user defined type which is very similar to a class in other languages. A *frame* contains as many variables or function definitions as necessary.

Variable Declaration

A variable is declared as in many language by giving first the type of the variable, then a list of variable names, separated by commas and ending with a “;”.

Example:

```
//each variable can be individually instantiated in the list
int i,j,k=10;
string s1="s1",s2="s2";
```

private type name;

A variable can be declared as *private*, which forbids its access from external sources: `private test toto;`

6 First Program

Since an example is better than a hundred lines of explanation, here is a small program, which simply displays the content of a string

```
//declaration
string s;
int i;

//Instantiation
s="abcd";
i=100;
//Display
print("S=",s," I=",i,"\n");
```

Execution

```
S=abcd I=100
```

7 Function, autorun, thread

A function is declared with the keyword *function*, followed with its name and parameters. A value can be returned with the keyword *return. Parameters are always sent as values except if the type is self*. It should be noted that a function does not provide any types for its return value.

7.1 Enforcing Return Type

A function can impose a return type which is evaluated on the fly when a value is returned… The return type is declared after the argument lists with a "::":

```
function toto(int i) :: int {
   i+=10;
   return(i);
}
```

We impose for instance that this function return an "int".

7.2 autorun

An *autorun* function is automatically launched after its declaration. Autorun functions are only executed in the main file. If you have autorun functions in a file which is called from within another file, then these functions are not executed.

Note: *autorun* are useless in frames.

Example

```
autorun waitonred() {
  println("Loading:",_paths[1]);
}
```

7.3 thread

When a *thread* function is launched, it is executed into an independent system thread.

Example:

```
thread toto(int i) {
  i+=10;
  return(i);
}

int j=toto(10);
print("J="+j+"\n");
```

Execution:

```
J=20
```

protected thread

"protected" prevents two threads to access the same lines of code at the same time.

A *protected* thread sets a *lock* (see below) at the beginning of its launch, which is released once the function is executed. Thus, different calls to a protected function will be done in a sequence and not at the same time. *Protected* should be used for code that is not *reentrant.*

Example

```
//We implement a synchronized thread
protected thread launch(string n,int m) {
  int i;
  println(n);
  //we display all our values
  for (i=0;i<m;i++)
                print(i," ");
  println();
}

function principal() {
  //we launch our thread
  launch("Premier",2);
  launch("Second",4);
  println("End");
}

principal();
```

run:

```
End
Premier
0 1
Second
0 1 2
```

exclusive thread

Exclusive is very similar to *protected,* with one difference. In the case of *protected*, the protection is at the method level, while with *exclusive* it is at the *frame* level. In this sense, *exclusive* works exactly as *synchronized* in Java.

In the case of a *protected* function, only one thread can have access to this *method* at a time, while if a method is *exclusive,* only one thread can have access to the *frame object* at a time, which means that different threads can execute the same method if this method is executed within different instances.

In other words, in a *protected* thread, we use a lock that belongs to the method, while in an *exclusive* thread, we use a lock that belongs to the frame instance.

```
exclusive thread framethod(..) { lock(instanceid)…}
protected thread method(…) {lock(methodid)…}
```

Example

```
//This frame exposes two methods
frame disp {

    //exclusive
    exclusive thread edisplay(string s) {
        println("Exclusive:",s);
    }

    //protected
    protected thread pdisplay(string s) {
        println("Protected:",s);
    }
}

//We also implement a task frame
frame task {
    string s;
    //with a specific "disp" instance
    disp cx;

    function _initial(string x) {
        s=x;
    }

    //Then we propose three methods
    //We call our local instance with protected
    function pdisplay() {
        cx.pdisplay(s);
    }

    //We call our local instance with exclusive
    function edisplay() {
        cx.edisplay(s);
    }

    //we call the global instance with exclusive
    function display(disp c) {
        c.edisplay(s);
    }
}

//the common instance
disp c;
vector v;
int i;
string s="T";
for (i=0;i<100;i++) {
    s="T"+i;
    task t(s);
    v.push(t);
```

```
}

//In this case, the display will be ordered as protected is not reentrant
//only one pdisplay can run at a time
for (i=0;i<100;i++)
   v[i].pdisplay();

//In this case, the display will be a mix of all edisplay working in parallel
//since, exclusive only protects methods within one instance, and we have different
//instances in this case...
for (i=0;i<100;i++)
   v[i].edisplay();

//In this last case, we have one single common object "disp" shared by all "task"
//The display will be again ordered as with protected, since this time we run into the same
// "c disp" instance.
for (i=0;i<100;i++)
   v[i].display(c);
```

joined and waitonjoined

A thread can be declared as *joined*, if the main thread is supposed to wait for the completion of all the threads that were launched before completing its own code, you can use waitonjoined*()* which will then wait for these threads to finish.

You can use as many *waitonjoined()* as necessary in different threads. *waitonjoined* only waits on "*joined threads"* that were launched within a given thread.

Atomic Values

Most data structures (maps, vectors, strings etc.) are protected in threads against concurrent access through a lock. However, these locks are often costly in terms of time and space. For instance, when many threads try to access the same data structure, the system will only guaranty the access to one thread at a time. The other threads will be put in hold, which means that their context will be saved in memory to be later reactivated when the lock is released. Tamgu provides some specific lock-free data structures (or *atomic types*) such as *a_int, a_string or a_mapii (*see the section about atomic types below). These data structures allow for a much efficient use of the machine, since when accessing these types, the threads are not put in hold. However, their implementation makes them slower than other types in non-threading context. Their use is only useful when they are shared across threads.

Stream Operator '<<<'

When you launch a thread, the result of that thread cannot be directly stored in a variable with the operator "=", since a thread lives its own life without any links to the calling code. Tamgu provides a specific operator for this task: <<<, also called the stream operator. A stream is a variable which is connected to the thread in such a way that the values returned by the thread can be stored within that variable. Of course, the variable must exist independently from the thread.

Example

```
//we create a thread as a "join" thread, in order to be able to use waitonjoined.
//This thread simply returns 2xi
joined thread Test(int i) {
    return(i*2);
}



//Our launch function, which will launch 10 threads
function launch() {
    //we first declare within this function our map storage variable
    treemapi m;
    int i=0;
    //we then launch 10 threads, and we store the result of each into m at a
specific position
    for (i in <0,10,1>)
        m[i]<<<Test(i);

    //we wait for all threads to finish
    waitonjoined();
    //we display our final value:
    println(m); //{0:0,1:2,2:4,3:6,4:8,5:10,6:12,7:14,8:16,9:18}
}

launch();
```

7.4 Multiple definitions

Tamgu allows for multiple definitions of functions sharing the same name, however differing in their parameter definition. For instance, one can implement a *display(string s)* and a *display(int s).*

In this case, when more than one function is implemented sharing the same name, the argument control is much stricter than with one single implementation as the system will try to find, which function is the most suitable according to the argument list of the function call. Thus, the mechanism through which arguments are translated into a value suitable for a function parameter is no longer available.

Example:

```
function testmultipledeclaration(string s, string v) {
   println("String:",s,v);
}
function testmultipledeclaration(int s, int v) {
   println("Int:",s,v);
}
//we declare our variables
int i;
int j;
string s1="s1";
string s2="s2";
//In this case, the system will choose the right function according to its argument list...
testmultipledeclaration(s1,s2); //the string implementation
testmultipledeclaration(i,j); //the integer implementation
```

7.5 Default Arguments

Tamgu supplies a mechanism to declare default arguments in a function. You can define for instance a value for a parameter, which then can be omitted from the call.

```
function acall(int x, int y=12, int z=30, int u=43) {
   println(x+y+z+u);
}

acall(10,5); //the result is: 88= 10+5+30+43
```

Note: Only the last parameters in a declaration list can be optional.

7.6 Specific flags: private & strict

Functions can also be declared with two specific flags that are inserted before the *function* keyword: *private and strict*.

Note:

If you wish to use both flags in the same definition, *private* should precede *strict.*

private [function | thread | autorun | polynomial]

When a function is declared *private*, then it cannot be seen from outside the current Tamgu file. If a Tamgu program is loaded from within another Tamgu program, *private* functions are unreachable from the loader.

Example:

```
//This function is invisible from external loaders…
private function toto(int i) {
    i+=10;
    return(i);
}
```

strict [function | thread | polynomial]

By default, when a function is declared in Tamgu, the language tries to convert each argument from the calling function into the parameters expected by the function implementation. However, this mechanism might be a bit too loose in certain cases when a stricter parameter checking is required. The *strict* flag helps solve this problem, since any function declared with this flag will demand strict parameter control.

Example:

```
strict function teststrictdeclaration(int s, int v) {
    println("Int:",s,v);
}

//we declare our variables
```

```
string s1="s1";
string s2="s2";

//In this case, the system will fail to find the right function for these parameters
and will return an error message…

teststrictdeclaration(s1,s2); //No string implementation
```

8 Loading external files: Type Tamgu vs *loadin*

The type *tamgu* is used to load a specific Tamgu program dynamically. *loadin* is used to load the content of a file into the current program.

8.1 Methods

1. **tamgu var(string Tamgupathname)**: *Create and load a Tamgu program*

8.2 Executing External Functions

The functions available in the Tamgu file can be called through a *tamgu* variable.

Example

```
//In our program test.tmg, we implement the function: Read

//In test.Tamgu
function Read(string s) {
   return(s+"_toto");
}
//------------------------------------------
//In call.Tamgu

//In our calling program, we first load test.Tamgu, then we execute Read

tamgu kf('c:\test.tmg'); //we load a program implementing Read
string s=kf.Read("xxx"); //we can execute Read in our local program.
```

private functions

If you do not want external programs to access specific functions, you can protect them by declaring these functions *private.*

Example

```
//we implement a function, which will cannot be called from outside
private function Cannotbecalled(string s) {…}
```

loadin(string pathname)

Tamgu also provides another way to load files in another program. *loadin(pathname)* loads the content of a file into the current program. In other words, *loadin* works exactly as if the current file did contain the code of what is loaded with *loadin*.

The code is actually loaded at the very place the call to *loadin* is actually written.

Example

```
//In our program test.tmg, we implement the function: Read

//In test.Tamgu
function Read(string s) {
            return(s+"_toto");
}
//-------------------------------------------
//In call.Tamgu

//In our calling program, we first load test.Tamgu, then we execute Read

loadin('c:\test.tmg'); //we load the code implementing Read
string s=Read("xxx"); //we can execute Read in our local program directly.
```

9 Frame

A frame is a class description which is used to declare complex data structures together with functions:

- Member variables can be instantiated within the frame.
- A method *_initial* can be declared, which will be automatically called for each new instance of this frame.
- A sub-frame is declared into the frame body. It automatically inherits the methods and variables from the top frame.
- Redefinition of a function is possible within a sub-frame.
- Private functions and variables can also be declared within a frame.

Example

```
frame myframe  {
  int i=10;         //every new frame will instantiate i with 10
  string s="initial";

  function display() {
             print("IN MYFRAME:"+s+"\n");
  }

  frame mysubframe  {
    function display() {
             print("IN MYSUBFRAME:"+s+"\n");
    }
  }
}
```

9.1 Using a frame

A frame object is declared with the name of its frame as a type.

Example

```
myframe first; //we create a first instance
mysubframe subfirst; //create a sub-frame instance

//We can recreate a new instance
first=myframe; //equivalent to "new myframe" in C++ or in Java

//To run a frame's function
myframe.display();
```

9.2 _*initial* function

A creator function can be associated to a frame. This function is automatically called when a new instance of that frame is created.

Example

```
frame myframe  {
  int i=10;         //every new frame will instantiate i with 10
  string s="initial";

  function _initial(int ij) {
       i=ij;
  }
  function display() {
       print("IN MYFRAME:"+s+"\n");
  }
}

// A new instance of myframe is created:
myframe second(10); //the parameters are then passed to the _initial function as in C++
```

9.3 _*final* function

The _*final* function is called whenever a frame object is deleted. Usually, an object which is declared in a function or in a loop is deleted once this function or this loop ends.

Important

- This function has no parameters.
- A call to that function does not delete the object.
- The content of this function cannot be debugged as it is called from within the destructor*, independently from the rest of the code.*

Example

```
frame myframe  {
   int i=10;         //every new frame will instantiate i with 10
   string s="initial";
```

```
        function _initial(int ij) {
            i=ij;
        }

        function _final() {
            print("IN MYFRAME:"+s+"\n");
        }
    }

    int i=10;
    while (i>=0) {
        // A new instance of myframe is created:
        //At the end of each iteration the _final function will be called
        myframe item(i);
        i--;
    }
```

9.4 Initialization Order

When items are declared within a frame, the call to the _*initial* function is done from the TOP down to its children.

Furthermore, if an item within a frame F is instantiated within the _*initial* function of that frame F, then this declaration takes precedence to any other declarations.

Example

```
//We declare two frames
frame within {
  int i;

  //with a specific constructor function
  function _initial(int j) {
  i=j*2;
  println("within _initial",i);
  }
}
```

```
//This frame declares a specific "within" frame
frame test {
   int i;
//In this case, we declare a specific frame, whose declaration depends on the variable i
  within w(i);

  //Our function _inital for that frame...
  function _initial(int k) {
  i=k;
  println("test _initial",k);
  }
}

//we create a test instance: t1 with as initial value: 20
test t1(20);
```

Execution

The execution yields the following result:

```
test _initial 20
within _initial 40
```

As we can see on this example, the *_initial function from test* was executed first. The call to *_initial* in *within*, was done after the execution, enabling the system to take advantage from the value of "*i*", which was declared in the frame description.

However, if one wants to initialize a *frame* element with a much more complex arrangement, it is possible to create the value from within the *_initial* function. In that case, any other declaration is useless.

Example

```
//This frame declares a specific "within" frame
frame test {
   int i;
   //In this case, we declare a specific frame, whose declaration depends on the
variable i
   within w(i);
```

```
    //Our function _inital for that frame...
    function _initial(int k) {
        i=k;
        //we replace the previous description with a new one
        //this declaration subsumes the other one above
        w=within(100);
        println("test _initial",k);
    }
}

//we create a test instance: t1 with as initial value: 20
test t1(20);
```

Execution

The execution yields the following result:

```
test _initial 20
within _initial 200
```

As we can see on this example, the explicit initialization of “w” in *_initial* replaces the declaration “*within w(i);*”, which becomes redundant.

9.5 Creation within the constructor

We have seen that it was possible to create a frame element by either declaring its initialization directly into the frame field list or within the constructor itself. When the frame element construction is made in the constructor, a simple declaration suffices; any other declaration would be redundant.

Example:

```
 //This frame declares a specific "within" frame
frame test {
    int i;
    //In this case, we postpone the actual creation of the element to the
constructor: _initial
    within w;
    //Our function _inital for that frame...
    function _initial(int k) {
        i=k;
        //we replace the previous description with a new one
        w=within(100);
        println("test _initial",k);
    }
}

//we create a test instance: t1 with as initial value: 20
test t1(20);
```

Important

If constructor parameters are required for “w”, and no creation of that element “w” is done in the constructor, then Tamgu will yield an error about missing parameters.

9.6 Common variables

Tamgu provides a very simple way to declare class variables. A class variable is a variable, whose value is shared across all instances of a given frame.

Example

```
frame myframe  {
    common int i;     //every frame will have access to the same common instance of that
variable.
}

myframe t1;
myframe t2;
t1.i=10;
t2.i=15;
println(t1.i,t2.i);  //display for both variables : 15 15
```

9.7 Private functions and members

Certain functions or variables can be declared as *private* in a frame. A *private* function or a *private* variable can only be accessed from within the frame.

Example

```
frame myframe  {
    int i=10;         //every new frame will instantiate i with 10
    //private variable
    private string s="initial";

    function _initial(int ij) {
        i=ij;
    }

    //private function
    private function modify(int x) {
        i=x;
        s="Modified with:"+x; //you can modify "s" here
    }

    function display() {
        modify(1000); //you can call "modify" here
        print("IN MYFRAME:"+s+"\n");
    }
}

myframe test;

//Illegal instructions on private frame members…
test.modify(100);  //this instruction is illegal as "modify" is private
println(test.s);       //this instruction is illegal as "s" is private
```

9.8 Sub-framing or enriching a frame

Tamgu enables the programmer to enrich or *sub-frame* an existing frame. A frame description can be implemented in a few steps. For instance, one can start a first description, then decides to enrich it later in the program.

Enriching

```
//We start with a limited definition of a frame…
frame myframe  {
   int i=10;         //every new frame will instantiate i with 10
}

//We add some code here after…
…

//Then we enrich this frame with some more code
//All we need is to use the same frame instruction as above, adding some new stuff

frame myframe  {
   function display() {
      println(i);
   }
}
```

Sub-frames

A sub-frame is a frame, which is declared within another frame (the parent frame), from which it inherits both variables and functions. A sub-frame can then replace functions from the parent frame with new versions and adds its own variables.

```
//If we want to add some sub-frames…
frame myframe  {
   //We can now add our sub-frame…
   frame subframe {…}
}
```

Using upper definition: frame::function

If you need to use the definition of the parent frame, instead of the current thread, Tamgu provides a mechanism, which is very similar to other languages such as C++ or Java. The function name must be preceded with the frame name together with "::".

Example

```
//Calling subframes...

//We define a test frame, in which we define a subtest frame
frame   test {
    int i;

    function _initial(int k) {
        i=k;
    }

    function display() {
        println("In test",i);
    }

    frame subtest {
        string x;

        function display() {
            println("In subtest",i);
            test::display();//will call the other display definition from test
        }
    }
}

//We create two objects
test t(1);
subtest st(2);

//We then call the different methods
t.display(); //display:"In test,1"
st.display();//display"In subtest,2" and "In test,2"
st.test::display(); //display "In test,2"
```

9.9 System Functions

Systems functions are used to map the different operators of the Tamgu language onto frame methods.

Comparison Functions

Tamgu also provides a way to help define specific comparison functions between different frame elements. These functions have a specific name, even though they will be triggered by the following operators: “>”,”<”,”==”,”!=”, “<=” and “>=”.

Each function has one single parameter which is compared with the current element.

Below is a list of these functions:

1.	equal:	function ==(frame b);
2.	different:	function !=(frame b);
3.	inferior:	function <(frame b);
4.	superior:	function >(frame b);
5.	inferior equal:	function <=(frame b);
6.	surperior equal:	function >=(frame b);

Each of these functions should return *true* or *false* according to their test.

Example:

```
//implementation of a comparison operator in a frame
frame comp {
    int k;
    //we implement the inferior operator
    function <(autre b) {
        if (k<b)
            return(true);
        return(false);
    }
}

//we create two elements
comp one;
comp two;
//one is 10 and two is 20
one.k=10; two.k=20;
//one is inferior to two and the inferior method above is called
if (one < two)
    println(“OK”);
```

Arithmetic functions

Tamgu provides also a mechanism to implement specific functions for the different numerical operators. These functions must have two operators, except for ++ and --. They must return an element of the same frame as its arguments.

1.	plus:	function +(frame a, frame b);
2.	minus:	function -(frame a, frame b);
3.	multiply:	function *(frame a, frame b);
4.	divide:	function /(frame a, frame b);
5.	power:	function ^^(frame a, frame b);
6.	shift left:	function <<(frame a, frame b);
7.	shift right:	function >>(frame a, frame b);
8.	mod:	function %(frame a, frame b);
9.	or:	function |(frame a,frame b);
10.	xor:	function ^(frame a,frame b);
11.	and:	function &(frame a,frame b);
12.	"++":	function ++();
13.	"--":	function --();

Example:

```
frame test {
    int k;

    function ++() {
        k++;
    }

    //it is important to create a new value, which is returned by the function
    function +(test a,test b) {
        test res;
        res.k=a.k+b.k;
        return(res);
    }
}
test a,b,c;
c=a+b; //we will then call our plus implementation above.
```

Number and string functions

You can also interpret a frame object as a string or a number in an arithmetic expression for instance. In that case, you can implement functions with as a name, the type of the object you want your frame instance to be interpreted as.

Example:

```
frame test {
    int k;
    string s;

    //We return a "test" instance as a string…
    function string() {
        return(s);
    }

    //We return a "test" instance as a float…
    function float() {
        return(k);
    }

    //We return a "test" instance as a int…
    function int() {
        return(k);
    }
}
```

Nota bene: In the case of numbers, you should note that you can define as many functions as the number of available types: *short, int, long, decimal* and *float*. However, *the first function to be defined will be the default numeric function for all types, unless you specifically add new function implementations.* Hence, in our example, all numerical type interpretation will be a *float* by default except for *int*.

Interval and index

It is also possible to use a frame object as a vector or a map. It is then possible to access elements through an interval or set a value through an index. To use an object in this way, the developer must implement the following function:

1. function [](self idx,self value): This function inserts an element in a vector at position idx
2. function [](self idx): This function returns the value at position idx.
3. function [:](self left,self right): This function returns the values between the position left and right.

Example:

```
frame myvect {
    vector kj;

    //This function inserts a value in the vector at position idx
    function [](int idx,self value) {
        kj[idx]=value;
    }

    //This function returns the value at position idx
    function [](int idx) {
        return(kj[idx]);
    }

    //This function returns the value between l and r.
    function [:](int l,int r) {
        return(kj[l:r]);
    }

}

myvect test;
test[0]=10;        //we call function [](…)
test[1]=5;         //we call function [](…)

//we call function [:](…)
println(test[0],test[1],test[0:]);      //Display: 10 5 [10,5]
```

10 Extensions

It is possible to extend some types, such as *map, vector, int, string and others* with new methods.

The notion of “extension” is very similar to the one of frame, except that the extension name should be one of the following types:

Valid types: string, automaton, date, time, file, integer, float, vector, list, map, set.

10.1 Extension definition

You define an extension as a frame, in which you declare the new methods you want your system to use.
If you need to refer to the current element, then you use a variable whose name is the type itself with “_” as a prefix.

For “extension vector”, then the variable will be: _*vector*.

Be careful, if you add new methods to “map”, then all maps will inherit these new methods. The same applies to vectors.

Example:

```
//we extend map to return a value, which will be removed from the map.
extension map {
   //we add this new method, which will be available to all maps...
   function returnandelete(int key) {
      //we extract the value with our key
      string s=_map[key];
      //we remove it
      _map.pop(key);
      return(s);
   }
}
map mx={1:2,3:4};
//returnandelete is now available to all types of map.
string s=mx.returnandelete(1);
imap imx={1:2,3:4};
```

```
int x=imx.returnandelete(1);
```

11 Tamgu Contextual

11.1 Tamgu is a contextual programming language.

The way a variable is *handled* depends on its *context* of utilization. Thus, when two variables are used together through an operator, the result of the operation depends on the type of the *variable on the left*, the one that introduces the operation. In the case of an assignation, the type of the receiving variable decides on the type of the whole group.

Example

If we declare two variables, one *string* and one *integer*, then the "+" operator will act as a concatenation or an arithmetic operation.

a) For instance, in this case, i is the receiving variable, making the whole instruction an arithmetic operation.

```
int i=10;
string s="12";
i=s+i;      //the s is automatically converted into a number.
print("I="+i+"\n");
```

Run

```
I=22
```

b) In this other case, s is the receiving variable. The operation is now a concatenation:

```
int i=10;
string s="12";
s=s+i;      //the i is automatically converted into a string.
print("S="+s+"\n");
```

Run

```
S=1210
```

11.2 Implicit conversion

This notion of context is very important as it defines how each variable should be interpreted. Implicit conversions are processed automatically for a certain number of types. For instance, an integer is automatically transformed into a string, with as value its own digits. In the case of a string, the content is transformed into a number if the string only contains digits, otherwise it is 0.

For more specific cases, such as a vector or a map, then the implicit conversions are sometimes a bit more complex. For instance, a vector as an integer will return its size and as a string a representation of this vector. A file as a string returns its filename and as an integer, its size in bytes.

12 Predefined Types

Tamgu provides many different objects, each with a specific set of methods. Each of these objects comes with a list of predefined methods.

12.1 Basic methods

All the types below share the same basic methods:

a) **isa(typename)**: *check if a variable has the type: typename (as a string)*

b) **type()**: *return the type of a variable as a string.*

c) **methods()**: *return the list of methods available for a variable according to its type.*

d) **info(string name)**: *return a help about a specific method.*

e) **json()**: *return the JSON representation of the object, when available.*

12.2 Transparent Object: *self* (ou *let)*

self is a transparent object, similar to a sort of pointer, which does not require any specific transformation for the parameter, when used in a function.

Note: The keyword *let* behaves as *self, with one big difference.* The first variable, which is stored in a *let* variable defines the type of that variable. In other words, if you store a *string* into a *let* variable, then this variable will always behaves as a string. You can modify this behaviour with the operator ":=", which in this case forces the *let* variable to a new type.

Example

```
function compare(self x, self y) {
    if (x.type()==y.type())
        print("This is the same sort of objects");
}
```

```
//For instance, in this case, the function compare receives two parameters, whose types might vary. A self declaration removes the necessity to apply any specific conversion to the objects that are passed to that function.
string s1,s2;
compare(s1,s2);

//we compare two frames
myframe i1;
myframe i2;
compare(i1,i2);
```

Example with *let*

```
let l="x=";

l+=12;
println(l); //it displays: x=12

//we force 'l' to be an integer
l := 1;

l+=12;
println(l); //it displays: 13
```

13 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 ©
```

13.1 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*

13.2 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.

	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]Englis

Part 1		h, 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:	C*eltic*
• e_latin_ffe= 15:	*Extended (French, Finnish, Estonian)*
• e_latin_see= 16:	S*outh 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']]];
```

13.3 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
```

14 Type Tamgu Regular Expression: treg

Tamgu provides its regular expression formalism, which is called: *treg.*

treg is both a type a specific way to write down regular expressions.
As a type, it takes as argument a string, which follows the description below. But it can also be provided directly as string, in that case this string is of the form: *r"…",* the '*r*' stands for regular expressions.

14.1 Methods

The type *treg* exposes two methods:

1. **compile(string rgx):** *compile a string into a treg*
2. **match(string s):** *check if the string s matches the Tamgu regular expression.*

Example:

```
treg tt("%C+");

if (tt.match("ABCDE"))
   println("Yes");

if (tt == "aABCDE")
   println("Yes");
else
   println("NO");
```

***r* expression**

You can also use *r* strings to do the same type of operation:

```
if (r"%C+" == "ABCDE")
   println("Yes");
```

An "r" expression can be written with "" or ''.
(Double quotes or simple quotes).

14.2 Language description

A Tamgu regular expression is a string where meta-characters are used to introduce a certain freedom in the description of a word. These meta-characters are the following:

%d	stands for any digit
%x	stands for a hexadecimal digit (abcdef0123456789ABCDEF)
%p	stands for any punctuation belonging to the following set: < > { } [ ]) , ; : . & | ! / \ = ~ # @ ^ ? + - * $ % ' _ ¬ £ €` “
%c	stands for any lower case letter
%C	stands for any upper case letter
%a	stands for any letter
?	Stands for any character
%?	Stand for the character “?” itself
%%	Stand for the character “%” itself
%s	stand for any space character include the non-breaking space
%r	stand a carriage return
%n	stand for a non-breaking space
~	negation
\x	escape character
\ddd	character code across 3 digits exactly
\xFFFF	character code across 4 hexas exactly
{a-z}	between a and z included
[a-z]	sequence of characters
^	the expression should start at the beginning of the string
$	the expression should match up to the end of the string

Example:

dog%c	matches *dogs* or *dogg*
m%d	matches m0, m1,…,m9

Operators: *,+, () , ([])

A regular expression can use the Kleene-star convention to define characters that occurs more than once.

x*:	the character can be repeated 0 or n times
x+:	the character must be present at least once
(x):	the character is optional
[xyz](+*):	A sequence of characters
{xyz}(+*):	A disjunction of characters

where x is a character or a meta-character. There is one special case with the '*' and the '+'. If the character that is to be repeated can be any character, then one should use **"%+"** or **"%*"** .

Important

These two rules are also equivalent to "?*" or "?+".

Example:

1) **a*ed** matches aed, aaed, aaaed etc. the *a* can be present 0 or n times)

2) a%*ed matches aed, aued, auaed, aubased etc. any characters can occur between *a* and *ed*)

3) a%d* matches a, a1, a23, a45, a765735 etc.

4) a{%d%p} matches a1, a/, a etc.

5) a{bef} matches ab, ae or af.

6) a{%dbef} matches a1, ab, ae, af, a0, a9 etc.

7) a{be}+ matches ab, ae, abb, abe, abbbe, aeeeb etc.

8) a[be]+ matches abe, abebe etc.

14.3 Macros: grammar_macros(key, pattern)

Some expressions might be a bit complex to write down. Tamgu provides the procedure *grammar_macros,* which creates a new meta-character, which can be used in expressions. The first argument is a character, which will be used as index, while the second is a regular expression pattern, that will be associated to this key.

This function provides patterns for all calls to grammar regular expressions.

Example:

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

string s="ceci est: 0x1.0d0e44bfeec9p-3 dans la chaine.";

//We use the macro
string res=s.scan("%X");

println(res); → 0x1.0d0e44bfeec9p-3
```

IMPORTANT

grammar_macros is a system function, which means that it is executed while parsing the code. It won't execute with the rest of code, once the compiling has been done.

14.4 Using *treg* in strings

It is also possible to use *treg* directly into strings to extract or modify their content.

```
string uu="That was: 0x1.0d0e44bfeec9p-3, 0x3.456aebp-1 in here.";

print("Tst:<",uu[-r"%d":" "],">\n");  //Tst:<1 >
println("Tst2:", uu["x":r"%d %c"]); //Tst2: x1.0d0e44bfeec9p-3, 0x3.456aebp-1 i
```

Note that it is also possible to replace a *treg* expression with a variable of type *treg*…

```
treg subt(r"%d %c");
println("Tst3:", uu["x":subt]); //Tst3: x1.0d0e44bfeec9p-3, 0x3.456aebp-1 i
```

taskell

You can also use these *treg* expressions with Taskell functions:

```
<mytreg(r"%C+") = "uppercase">
<mytreg([r"%d+":v]) = "ok">
<mytreg(v) = "missed">

println(mytreg("ABC")); //uppercase
println(mytreg(["3a4",1,4,5])); //ok
```

Prolog

You can also use them in Prolog…

```
tst(r"%d+"). // ← the treg expression is here…
tst("machin").

truc("Ok", ?P) :- tst(?P).

vector vpred;
vpred = truc(?P,"machin");
println(vpred); //[truc("Ok",machin)]

vpred = truc(?P, "12");
```

```
println(vpred); //[truc("Ok",12)]

vpred = truc(?P, "nope");
println(vpred); //[]
```

14.5 Posix Regular Expressions: preg

The posix regular expression are also available in Tamgu, in the same way as Tamgu Regular expression.

However, there are two minor differences. First, to use these expressions you have to prefix your expression with: *p*, *instead of r.*

Second, the object type associated is: *preg.*

These regular expressions are based on the posix regular expression scheme. They can be used instead of treg everywhere, except for the *scan* method, in strings and files.

Example:

```
string str="this subject has a submarine as a subsequence";

svector vs = p'\b(sub)([^ ]*)' in str;
println(vs); // ['subject','submarine','subsequence']


ivector iv = p'\b(sub)([^ ]*)' in str;
println(iv); // [5,12,19,28,34,45]


string s = p'\b(sub)([^ ]*)' in str;
println(s); //subject

s = str[p"\b(sub)([^ ]*)"];
println("S="+s); // S=subject


s = str[-p"\b(sub)([^ ]*)"];
println("S="+s); // S=subsequence
```

15 Type: byte, short, int, float, real, long

Tamgu provides different numerical types: *byte, short, int, float, real* and *fraction*, which is described in the next section.

N.B. *real* and *float* are an alias of one another. *real* was added because *float* was often misunderstood as was it stood for.

Note about the C++ implementation:

int and *float* (or *real*) have been implemented respectively as a *long* and a *double. long* is implemented as a 64 bits integer, respectively a __int64 on Windows or a "long long" on Unix platforms.
byte is implemented as a *unsigned char* and *short* is implemented as *short.*

15.1 Methods:

1. **#():** *Return the byte complement*
2. **abs():** *absolute value.*
3. **acos():** *arc cosine.*
4. **acosh():** *area hyperbolic cosine.*
5. **asin():** *arc sine.*
6. **asinh():** *area hyperbolic sine.*
7. **atan():** *arc tangent.*
8. **atanh():** *area hyperbolic tangent.*
9. **base(int b):** *return a string representing a number in base b*
10. **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 set of characters with your own. If you supply an empty vector, then the system resets to the default set of characters.*
11. **bit(int ith):** *return true, if the ith bit is 1.*
12. **cbrt():** *cubic root***.**

13. **chr():** *return the ascii character corresponding to this number as a code.*
14. **cos():** *cosine.*
15. **cosh():** *hyperbolic cosine.*
16. **emoji():** *return the textual description (in English) of an emoji based on its Unicode code.*
17. **erf():** *error function.*
18. **erfc():** *complementary error function.*
19. **even():** *return true if the value is even.*
20. **exp():** *exponential function.*
21. **exp2():** *binary exponential function.*
22. **expm1():** *exponential minus one.*
23. **factors():** *return the prime factor decomposition as an ivector.*
24. **floor():** *down value.*
25. **format(string form):** *return a string formatted according to the pattern in form. (this format is the same as the sprintf format in C++)*
26. **fraction():** *return the value as a fraction.*
27. **isemoji()***: return true if the code matches an emoji.*
28. **lgamma()***: log-gamma function.*
29. **log()***: natural logarithm.*
30. **log1p()***: logarithm plus one.*
31. **log2()***: binary logarithm.*
32. **logb()***: floating-point base logarithm.*
33. **nearbyint()***: to nearby integral value.*
34. **odd()***: return true if the value is odd.*
35. **prime()***: return true is the value is a prime number.*

36. **rint()**: *to integral value.*

37. **round()**: *to nearest.*

38. **sin()**: *sine.*

39. **sinh()**: *hyperbolic sine.*

40. **sqrt()**: *square root.*

41. **tan()**: *tangent.*

42. **tanh()**: *hyperbolic tangent.*

43. **tgamma()**: *gamma function.*

44. **trunc()**: *value.*

Specific to floating point values (float and decimal)

45. **bits() :** *return the underlying bit representation of a floating point value as an integer (int or long)*

46. **bits(int v):** *transforms v into a floating point value. v in this case not an integer value, but the inner bit representation of a floating point value, such as the one returned by* bits().

47. **exponent():** *return the exponent of a floating point value, such as:* value = mantissa()*2^^exponent().

48. **mantissa():** *return the mantissa of a floating point value.*

Complete list of mathematical functions

Tamgu provides the following mathematical functions:

abs, acos, acosh, asin, asinh, atan, atanh, cbrt, cos, cosh, erf, erfc, exp, exp2, expm1, floor, lgamma, ln, log, log1p, log2, logb, nearbyint, rint, round, sin, sinh, sqrt, tan, tanh, tgamma, trunc.

Hexadecimal

A hexadecimal number always starts with “0x”. It is considered by Tamgu as a valid number as long as it is a valid hexadecimal string. A hexadecimal declaration can mix upper or lower characters for the hexadecimal digits: A,B,C,D,E,F.

Operators

+,-,*,/:	mathematical operators
<<,>>,&,|,^:	bitwise operators
%:	division modulo
^^:	power (2^^2=4)
+=,-= etc:	self operators

Syntactic Sugar

Tamgu provides some syntactic sugar notations, which makes operations a little more readable.

- ×,÷ can be used instead of * and /
- π, τ, φ,ℯ, whose values are 3.14159, 6.2831, 1.61803 and 2.71828
- _pi, _tau, _phi, _e are another name for the values above
- ²,³ for square and cubic…
- √(,∛(for square root and cubic root.
- You can also write expression such as: 2a+b or 2(12+a)
- a b (with a space in between) is the same as: a*b or a×b

Example

```
int h = 0xAB45;         //Hexadecimal number
int i=10;
float f=i.log();   //value= 1
f+=10;             //value=11
f=i%5;             //value=0
f=2i+10;           //30
f=2×i+10;          //30
f=2π+φ;            //7.90122
f=√(2i);           //4.47214
f=i²;              //100
f=2(i-1);          //18
```

16 Type iloop, floop, bloop, sloop, uloop

These two types are used to define looping variables. A looping variable is a variable whose value evolves in an interval. They are initialized with a vector definition and each time a “++” is called upon them, they jump to the next value. When they reach the end of the interval, they start all over again at the beginning.

- *iloop* loops among *integer*
- *floop* loops among *float*
- *bloop* loops between *true* and *false*.
- *sloop* loops among *string*
- uloop loops among ustring

16.1 Initialization

You initialize a *loop* with a vector or a range.

```
iloop il=[1,3..10];
```

For instance, in the example above, the variable *il* will loop between the values *1,3,5,7,9*.

With an integer

If you initialize a *loop* with an integer, then this value will be considered as ***a new position into the associated vector*.**
The value 0 resets the loop to the first element. The value -1 resets the loop to the last element.

```
il=3; //the variable is now 7. The next value will be 9.
```

16.2 As a Vector

You can return the vector value of a *loop*, with the method *vector* or by storing its content into a vector.

16.3 Function

You can also associate a function to a *loop* variable, which will be called *when the last value of the initial vector is reached before looping again.* The function exposes the following signature:

```
function callback(loop var,int pos);
```

Examples:

```
iloop i=[1..4];

for (int k in <0,10,1>) {
    print(i," ");
    i++;
}
```

The system prints: **1 2 3 4 1 2 3 4 1 2**

17 Type fraction

Tamgu enables users to handle numbers as fractions, which can be used anywhere in any calculations. All the above mathematical methods for integers and floats are still valid; however this type offers a few other specific methods.

17.1 Methods:

1. **d()**: *return the denominator of the fraction*
2. **d(int v)**: *set the denominator of the fraction*
3. **fraction f(int n,int d)**: *a fraction can be created with a numerator and a denominator. By default, the numerator is 0 and the denominator is 1.*
4. **invert()**: *switch the denominator with the numerator of a fraction*
5. **n()**: *return the numerator of the fraction*
6. **n(int v)**: *set the numerator of the fraction*
7. **nd(int n,int d)**: *set the numerator and denominator of a fraction*

As a string, an integer or a float

Tamgu automatically creates the appropriate float or integer, through a simple computing of the fraction. This translation results in most of the cases into a loss of information. Furthermore, at each step, Tamgu simplifies the fraction in order to keep it as small as possible.

As a string, Tamgu returns: “NUM/DEN”

Examples:

```
//we create two fractions
fraction f(10,3);
fraction g(18,10);
//we add g to f...
f+=g;
println(f); //Display: 77/15
```

18 Type vector

A vector is used to store any objects, whatever their type. It exposes the following methods.

18.1 Methods

1. **clear()**: *clear the container.*
2. **convert():** *detect number values in string vectors (svector and uvector only) and convert them into actual numbers. Return a vector object.*
3. **editdistance(v)**: *Compute the edit distance with vector 'v'.*
4. **flatten()**: *flatten a vector structure.*
5. **insert(i,v)**: *Insert v at position i.*
6. **join(string sep)**: *Produce a string representation for the container.*
7. **json()**: *return a json compatible string matching the container.*
8. **last()**: *return the last element.*
9. **merge(v)**: *Merge v into the vector.*
10. **pop(i)**: *Erase an element from the vector*
11. **product()**: *return the product of elements.*
12. **∏(v,i,j)** *: multiply the elements from i to j, i,j are optional.*
13. **push(v)**: *Push a value into the vector.*
14. **read(string path):** *read a file into the vector (vector, svector, uvector only)*
15. **remove(e)**: *remove 'e' from the vector.*
16. **reserve(int sz)**: *Reserve a size of 'sz' potential element in the vector.*
17. **reverse()**: *reverse a vector.*
18. **size()** *: Number of elements, size of the container.*
19. **shuffle()**: *shuffle the values in the vector.*

20. **sort(bool reverse | function | method)**: *sort the values in the vector.*
21. **sortfloat(bool reverse)**: *sort the values in the vector as float.*
22. **sortint(bool reverse | function)**: *sort the values in the vector as int.*
23. **sortstring(bool reverse)**: *sort the values in the vector as string.*
24. **sortustring(bool reverse | function)**: *sort the values in the vector as ustring.*
25. **sum()**: *return the sum of elements.*
26. **∑(v,i,j)**: *sum the elements from i to j, i,j are optional.*
27. **unique()**: *remove duplicate elements.*
28. **write(string file):** *write the vector content into a file*

18.2 Initialization

A vector can be initialised with a structure between "[]".

```
vector v=[1,2,3,4,5];
vector vs=["a","b","v"];
vector vr=range(10,20,2); // vr is initialized with [10,12,14,16,18];
vs=range('a','z',2); //vr is initialized with ['a','c','e','g','i','k','m','o','q','s','u','w','y']
```

18.3 Mathematical functions

You can also apply a mathematical function onto the content of a vector. See the numerical types (*int, float, long*) for a list of these functions:

Example:

```
fvector fv=[0,01..1];
fv is: [0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1]
```

fv.cos() is:

[1,0.995004,0.980067,0.955336,0.921061,0.877583,0.825336,0.764842,0.696707,0.62161,0.540302]

Operators

x in vect: return true or a list of indexes, according to the receiving variable. If the vector contains strings, then the system will return true or its index, only if the value is the same string as the one tested. A in is not PERFORMED in this case within the local strings.

for (s in vect) {…}: loop among all values. At each iteration "s" contains a value from vect.

+,*,-,/ etc..: add etc.. a value to each element of a vector or add each element of a vector to another

&,|: intersection or union of two vectors

&&&: merge a vector with a value

||| : combine a container with another container, or a string with a string.

```
vector v= [1,2,3];
vector vv = [4,5,6];
println(v ||| vv); //[[1,4],[2,5],[3,6]]
```

:: : insert a value in a vector.

10::[1,2,3] → [10,1,2,3]

[1,2,3]::10 → [1,2,3,10]

As an integer or a Float

It returns the size of the vector

As a string

It returns a structure, where each element is separated from the others with a comma, similar to the structure used to initialize a vector.

Indexes

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

str[i:j]: return the sub-vector between i and j.

Extracting variables from a vector

Tamgu provides a very peculiar method to read from a vector. You can use a vector pattern of the form: [a_1,..,a_n|tail], where a_1,..,a_n, tail are variables or values. The tail is the rest of the vector, once each variable has been assigned.

These vector patterns can be used in two ways:

- In assignment:
  - [a,b|v]=[1,2,3,4,5], then a=1, b=2 and v=[3,4,5]
- In *for..in* loops
  - for ([a,b|v] in [[1,2,3,4],[3,4,5]]) etc…

In the first iteration, a=1,b=2 and v=[3,4]

In the second iteration, a=3,b=4 and v=[5]

Example

```
vector vect;

vect=[1,2,3,4,5];
print(vect[0]);             //display: 1
print(vect[0:3]);           //display: [1,2,3]
vect.push(6);
print(vect);                //display: [1,2,3,4,5,6]
vect.pop(1);
print(vect);                //display: [1,3,4,5,6]
vect=vect.reverse();
print(vect);                //display:[6,5,4,3,1]
vect.pop();
print(vect);                // display:[6,5,4,3]
vect+=10;
print(vect);                // display:[16,15,14,13]
```

Example (sorting out integers in a vector)

```
//This function should return only true or false
```

```
//The type of the parameters will determine its behaviour, in this case, we
//suppose each element to be a string or converted as a string.

function compare(int i,int j) {
   if (i<j)
      return(true);
   return(false);
}

vector myvect=[10,5,20];
myvect.sort(compare);
```

Result is: [5,10,20]…

Example (sorting out integers in a vector but seen as strings)

```
function compare(string i,string j) {
   if (i<j)
      return(true);
   return(false);
}

vector myvect=[10,5,20];
myvect.sort(compare);
```

Result is: [10,20,5]…

Example (sorting out strings with the 'size' method)

```
vector v= ["100","1","22"];
v.sort(size);
```

Result is: ['1','22','100']…

19 Type list

A list is used to store any objects, whatever their type. It exposes the following methods. It is different from *vector* in the sense that it works as a list in which, elements can added at the front or at the back, and can be removed from the front and from the back, allowing FIFO, LILO, FILO, or LIFO management of lists.

19.1 Methods

1. **clear()**: *clear the container.*
2. **first()**: *return the first element.*
3. **flatten()**: *flatten a vector structure.*
4. **insert(i,v)**: *Insert v at position i.*
5. **join(string sep)**: *Produce a string representation for the container.*
6. **json()**: *return a json compatible string matching the container.*
7. **last()**: *return the last element.*
8. **merge(v)**: *Merge v into the list.*
9. **pop(i)**: *Erase an element from the list at position i.*
10. **popfirst(i)**: *remove and return the first element.*
11. **poplast(i)**: *remove and return the last element.*
12. **product()**: *return the product of elements.*
13. **push(v)**: *Push a value into the list.*
14. **pushfirst(v)**: *Push a value into the list in first position.*
15. **remove(e)**: *remove 'e' from the vector.*
16. **reverse()**: *reverse a vector.*
17. **size()** *: Number of elements, size of the container.*
18. **shuffle()**: *shuffle the values in the list.*

19. **sum()**: *return the sum of elements.*

20. **unique()**: *remove duplicate elements.*

Initialization

A list can be initialised with a structure between “[]”.

```
list v=[1,2,3,4,5];
list vs=["a","b","v"];
```

Operators

x in vlist: return true or a list of indexes, according to the receiving variable. If the list contains strings, then the system will return true or its index, only if the value is the same string as the one tested. A in is not PERFORMED in this case within the local strings.

for (s in vlist) {…}: loop among all values. At each iteration s contains a value from vlist.

+,*,-,/ etc..: add etc.. a value to each element of a list or add each element of a list to another

&,|: intersection or union of two lists

As an integer or a Float

It returns the size of the list

As a string

It returns a structure, where each element is separated from the others with a comma, similar to the structure used to initialize a vector or a list.

Indexes

You can use indexes with *list* objects, as with vector. However, indexes with lists are rather inefficient, and should be avoided.

Example

```
list vlist=[1,2,3,4,5];


vlist.pushfirst(10);
vlist.pushlast(20); //display: [10,1,2,3,4,5,20]
vlist.popfirst();//display: [1,2,3,4,5,20]

vector v=vlist; //transform a list into a vector
```

20 Type [b|i|f|s|u]vector, table

20.1 Type bvector, ivector, lvector, fvector, svector, uvector

These five types are specialized vector containers for bytes (*bvector),* integers (*ivector*), longs (*lvector),* floats (*fvector*), strings (*svector*), unicode strings (*uvector*).

These containers can only store their specific type of values. They are very useful to keep the memory consumption of these elements in check. Basically, when you store a string in a *vector*, Tamgu needs to create a *string* object, which will be stored within your *vector*, since a vector can only store *objects.* In the case of a *svector*, *the system will store the string directly without requesting Tamgu to create any specific string object.* The storage is then reduced to only strings and the access is both faster and leaner.

You use these structures exactly in the same way as a vector.

```
svector test;
test.push("toto");
```

20.2 Type table

The last type, “table”, is a container whose size must be defined at creation, once for all. It expects integers as indexes…

```
table test(10);
test[1]="i";
```

This container is extremely fast, as it is based on a C table implementation, however, its limitations are the ones set by its size at creation. However, if the initial size is too small, you still can use “resize” to enlarge or decrease that initial size.

```
table test(10);
println(test.size()); //10

test.resize(20);
println(test.size()); //the size is now 20
```

Not only will this method modify the current size of your table, it will also copy all previous elements in their new place. Note, that if you actually decrease the size of the table, elements beyond the new limit will be lost.

Important: this table is not protected for *read/write* in threads. If you can ensure that no simultaneous *read/write will occur on the same elements*, then this structure might be very efficient to use as it will reduce the number of internal locks. However, if you predict some potential collisions, it is safer to use locks to avoid crashes.

Furthermore, Tamgu prevents you from *resizing* within a thread, as the concurrent access to elements might be disrupted.

21 Type (tree, bin, prime)map

A map is a hash table, which uses as key any string or any element which can be analysed as a string. The map in Tamgu converts *any keys* into a string, which basically means that "123" and 123 are one and unique key.

Note:

treemap is similar to *map,* with a difference that keys in a *treemap* are automatically sorted out.

binmap is also similar to map, however keys are *short*, whose values are between 0 and 65535. Keys are always sorted out. *Binmap* is also the fastest way to access elements.

Note: *binmap* is available through the "*allmaps"* library (*see specialized map below*)

primemap is a hash-map, where keys are organized along prime numbers. The advantage of this map is that you can iterate along the order in which the values were stored in the map.

21.1 Methods

1. **clear()***: clear the container.*
2. **find(value)***: test if a value belongs to the map and return 'true' or the corresponding keys.*
3. **invert()***: return a map with key/value inverted.*
4. **items()***: Return a vector of {key:value} pairs.*
5. **join(string sepkey, string sepvalue)***: Produce a string representation for the container.*
6. **json()***: return a json compatible string matching the container.*
7. **keys()***: Return the map container keys as a vector.*
8. **merge(v)***: Merge v into the vector.*
9. **pop(key)***: Erase an element from the map*
10. **product()***: return the product of elements.*

11. **size()** *: Number of elements, size of the container.*
12. **sum()***: return the sum of elements.*
13. **test(key)***: Test if key belongs to the map container.*
14. **values()***: Return the map container values as a vector.*

Initialization

A map can be initialised with a description such as: {“k1”:v1,”k2”:v2...}

map toto= {“a”:1,”b”:2};

Operator

x in amap: return true or a list of indexes, according to the receiving variable. If the map contains string values, then the system will return true or its index, only if a value is the same string as the one tested. A **in** is not PERFORMED in this case within the local strings.

Important:

x is tested against the *keys* of the map as for *test*.

for (s in amap) {…}: loop among all keys. At each iteration “s” contains a key from amap.

+,*,-,/ etc..: add etc.. a value to each element of a map or add each element of a map to another along keys

&,|: intersection or union of two maps along keys.

Indexes

map[key]: return the element whose key is key. If key is not a key from map, then return null.

As an integer or a float

Return the size of the map

As a string

Return a string which mimics the map initialization structure.

Example

```
map vmap;

vmap["toto"]=1;
vmap[10]=27;

print(vmap);                //display: {'10':27,'toto':1}
```

21.2 Testing keys

There are different ways to test whether a map possesses a specific key. The first way is to use the *test* operator, which will return *true* or *false*. The other way is to catch the error when a wrong index is provided with the container.

However, it is faster and more efficient to use *test* instead of the above equality.

```
if (m.test("ee"))
   println("ee is not a key in m ");
```

if you want to avoid an exception whenever a wrong key is used, place *erroronkey(false)* at the beginning of your code. In that case, an *empty* value will be returned instead of an exception.

```
if (m["ee"]==empty)
   println("ee is not a key in m ");
```

22 Specialized maps

22.1 (tree|prime|bin)map[s|i|f|u|l]

These types are very similar to "map" and to "treemap" with one exception, they use *integer (mapi,treemapi,primemapi), float (mapf,treemapf,primemapf) or ustring (mapu, treemapu, primemapu)* as keys while "map", "treemap" and "primemap" use *strings.*

Actually, for consistency reason, map, treemap or primemap can also be named: maps, binmaps, treemaps and primemaps.

Important:

These maps can only be accessed through the *allmaps* libraries. If you want to use one of these extended maps, you need to load *allmaps* with *use* beforehand.

```
use("allmaps");

treemapll t;
```

However, some maps are already part of the main interpreter and do not require loading this library first. Here is the complete list:

1) all variations for *map*.
2) treemaps[s|f|i|l], treemapi, treemapis, treemapsf
3) primemaps[s|f|i|l]

For the others, you need to load "allmaps" first.

22.2 Specialized value maps.

These specific maps have a key, which can be a string, an integer or a float and a value which is necessary also a string, an integer or a float. The naming convention in this case is:

(tree|prime)map[s|i|f|u][s|i|f|u]

For instance, *treemapif* is a *treemap* whose key is an integer and the value a float.

These specialized maps should be used as much as possible when the values and keys are basic values. They reduce the memory footprint of applications in a rather important way and are much faster to work with.

23 Self-included containers

As we have seen above, there are basically two sorts of containers in Tamgu.

a) Containers that contain values (svector, ivector or mapii)
b) Containers that contain objects (vector, map, treemap, table or list)

We will call them respectively: value containers and object containers.

23.1 Loops in Object Containers

Nothing prevents you from including a container into another container. You can actually include a container into itself. When you include a container within itself, the system cannot display the whole structure and will replace the self-reference with: […] or {…} if it is a map:

Example:

```
vector v=[1..5];
v.push(v);

println(v);

v is: [1,2,3,4,5,[...]], the [...] indicates a self-reference
```

However, if a container contains itself, there might be a problem when traversing this structure as you might loop indefinitely.

23.2 Marking methods

Tamgu provides three methods to help you deal with this problem:

a) mark(byte v): this method put a mark on an object container
b) mark(): this method returns the mark on an object container
c) unmark(): this method unmarks all marks within an object container

Example:

```
vector v=[10..50:10];

v.push([60..80:10]);
```

```
v.push(v);
println(v);

function traversal(self v, int i) {

   if (i==v.size()) {
      println('End of vector');
      return;
   }

   if (v[i].mark()==1) {
      println("Already analyzed");
      return;
   }

   println(i,v[i]);
   v[i].mark(1);
   if (v[i].iscontainer()) {
      println("This value is a container...");
      traversal(v[i],0);
   }
   traversal(v,i+1);
}

//We mark our vector to avoid traversing it twice
v.mark(1);
traversal(v,0);
//We unmark all elements at once
v.unmark();

V is: [10,20,30,40,50,[60,70,80],[...]], it contains a self reference

//The run
0 10
1 20
2 30
3 40
4 50
5 [60,70,80]
This value is a container...
0 60
1 70
2 80
End of vector
Already analyzed
end
```

24 Lock free Types

Lock free types are very useful in multithreading applications. They provide a way to manipulate numbers, strings, vectors or maps without any locks. Locks have a heavy cost on execution. When a thread is put on hold on a lock, the system must store its context and re-activate it when the lock is released. Lock-free programming aims at removing locks as much as possible to avoid costly stops in the execution flow.

table and rawstring

We have seen in previous sections two of these lock-free objects: *table* and *rawstring.*

The specificity of these objects is that in the context of a thread their size is fixed and cannot be modified, which guaranties that any concurrent access to these objects will not hamper their inner buffer declaration.

a_bool, a_int, a_float, a_string, a_ustring

Tamgu provides also five atomic values for Boolean, integers, floats and strings. These objects can be used as regular values, however in case of concurrent access, their atomicity is guaranteed. Atomicity in this context means that only one thread can modify their value at a time.

a_vector, a_[i|f|s|u]vector, a_map[i|f|s|u][i|f|s|u]

Tamgu also provides containers, which can be modified by different threads in the same time with minimum locks. Again, these structures can be used exactly as their corresponding non-atomic maps. These structures are implemented as a linked list of chunks. Each chunk can usually accommodate quite a few values. The only time when a lock is set is when a new element is added to this linked list. Reading and most storage are processed without any locks, except for the case when the map or the vector is not large enough to accommodate a new element.

Note: *a_maps* are all value containers while *a_vector* is an object container.

Note bis: *you need to load the “allmaps” library to have access to these maps.*

ring

“ring” is also a lock-free container, which can contain at most 65535. *Ring* is implemented as structure in which elements can be stored or removed on the front or on the back. These operations, contrary to other containers, have exactly the same memory time or space print.

Important

When different threads try to modify the same lock-free variable, there is no guarantees that the two modifications will be both successful. The last to access the variable might be the one to modify the value.

25 Type fmatrix, imatrix

These types are used to handle matrices. You define the matrix size at creation time and you can store elements with a redefinition of the ":" operator. In this case, this operator is used to define the rows and columns of the value to store. Matrices can only store floats.

m[r:c]=v: we store an element v at row r and column c.

m[r:]	returns the row r as a fvector
m[:c]	returns the row c as a fvector

25.1 Methods

1. **determinant()**: *return the matrix determinant*
2. **dimension()**: *return the matrix size.*
3. **dimension(int rowsizeint columnsize)**: *set the matrix size.*
4. **invert()**: *Return the inverted matrix.*
5. **product()**: *return the product of all elements*
6. **sum()**: *return the sum of all elements*
7. **transpose()**: *return the transposed matrix*

Operators

The different operators: +,*,/,- are all available. However, note that the multiplication of two matrices multiplies two matrices one with other according to matrix multiplication. The same is true for division.

Examples

```
//We define the number of rows or columns
fmatrix m(3,3);
fmatrix v(3,1);

//We store elements,
v[0:0]=3;v[1:0]=0;v[2:0]=0;
```

```
float angle=56;

function loading(fmatrix mx,float θ) {
    θ=θ.radian();
    mx[0:0]=cos(θ);    mx[0:1]=0;    mx[0:2]=sin(θ);
    mx[1:0]=0;    mx[1:1]=1;    mx[1:2]=0;
    mx[2:0]=-sin(θ);    mx[2:1]=0;    mx[2:2]=cos(θ);
}


loading(m,angle);

fmatrix vx;
//Matrix multiplication
vx=m*v;

m[0:0]=-2;m[0:1]=2;m[0:2]=-3;
m[1:0]=-1;m[1:1]=1;m[1:2]=3;
m[2:0]=2;m[2:1]=0;m[2:2]=-1;

//The determinant
int det=m.determinant();
println(det);

m[0:0]=1;m[0:1]=2;m[0:2]=-1;
m[1:0]=-2;m[1:1]=1;m[1:2]=1;
m[2:0]=0;m[2:1]=3;m[2:2]=-3;

fmatrix inv;

//Matrix inversion
inv=m.invert();
```

26 Value Containers Logical Operators: &,|,^

The value containers are specific implementations of vectors and maps for strings, floats and integers. If you use logical operators with these containers, then the way they are processed depends on the values stored in the container.

For strings, the logical operators work as set operators. The & yields the intersection between two string containers, the | yields the union of two string containers, while the ^ yields the non-common values between two strings.

```
svector sv=["a","b","c","d",'e','h'];
svector svv=['e',"f","g",'h'];


println("And:",sv&svv);         → ['e','h']
println("XOR:",sv^svv);         → ['f','g','a','b','c','d']
println("OR:",sv|svv);          → ['a','b','c','d','e','h','f','g']

smap sm={"a":1,"b":2,"c":3,"d":4,'e':5,'h':6};
smap smm={'e':5,"f":2,"g":3,'h':4};

println("And:",sm&smm);         → {'e':'5'} 'h' has a different value…
println("XOR:",sm^smm);         → {'f':'2','g':'3','a':'1','b':'2','c':'3','d':'4'}
println("OR:",sm|smm);          → {'a':'1','b':'2','c':'3','d':'4','e':'5','h':'6','f':'2','g':'3'}
```

For numerical values, the logical operators work as the other operator at the binary level, not at the set level.

```
ivector iv=[1,2,3,4,5,6,7,8,9];
ivector vi=[2,4,6,8,10,12,14,16,18];

println("And:",iv&vi);  →      [0,0,2,0,0,4,6,0,0]
println("XOR:",iv^vi);  →      [3,6,5,12,15,10,9,24,27]
println("OR:",iv|vi);   →      [3,6,7,12,15,14,15,24,27]
```

27 Type transducer

This type is focused on storing and handling lexicons in a very compact and efficient way.

This type exposes the following methods:

27.1 Methods

1. **add(container,bool norm,int encoding)***: transform a container (vector or map) into a transducer lexicon. If the container is a vector, then it should have an even number of values.*

2. **build(string input,string output,bool norm,int encoding):** *Build a transducer file out of a text file containing on the first line surface form, then on next line lemma+features.*

3. **compilergx(string rgx,svector features,string filename):** *Build a transducer file out of regular expressions. filename is optional, the resulting automaton is stored in a file.*

4. **load(string file)***: load a transducer file.*

5. **lookdown(string lemFeat, byte lemma):** *Searching for a surface form, which matches a lemma plus features.* Lemma *is optional. When it is 1 or 2, then the string to compare against can be reduced to only a lemma. If lemma is 2 then features are also returned.*

   Important: The lemma should be separated from the features with a tab.

6. **lookup(string wrd, int threshold, int flags)**: *Lookup of a word using a transducer and a set of potential actions combined with a threshold. These two last arguments can be omitted.*

   a. **a_first:** *the automaton can apply actions to the first character. If this action is not set, then all the strings that will be compared against it will start with this character. If this character is not present in the automaton, then the system will switch to this mode.*

   b. **a_change:** *the automaton can change a character to another*

   c. **a_delete:** *the automaton can delete a character*

   d. **a_insert:** *the automaton can insert a character*

   e. **a_switch:** *the automaton transposes two characters*

   f. **a_nocase:** *the automaton checks if the two characters can match independently of their case.*

   g. **a_repetition:** *the automaton accepts that a character is repeated a few times.*

   h. **a_vowel:** *the automaton compares de-accentuated vowels together. For instance "e" will match "é" or "è" but not "a".*

   i. **a_surface:** *the automaton only returns surface form.*

7. **parse(string sentence, int option, int threshold, int flags):** *Analyse a sequence of words using a transducer. Option takes the following values:*

   a. *0: returns only the surface forms that were recognized within the initial string*

   b. *1: same as 0 with their offsets.*

   c. *2: return the surface forms and the lemmas with their features that were recognizes in the initial input.*

   d. *3: same as 2 with their offsets.*

   The threshold and the flags are optional. They follow the same convention as for lookup.

8. **store(string output,bool norm,int encoding):** *Store a transducer into a file. The last two parameters are optional*

27.2 Format

The format of files that are compiled into lexicons either through *build* or through *add*, have a similar structure.
In the case of a file, the first line should be a surface form, while the next line should be a lemma with some features, separated with a tab and so on so forth:

```
classes
class	+Plural+Noun
```

```
class
class    +Singular+Noun
etc.
```

The function *build* takes such a file as input and generates a file which contains the corresponding transducer out of these lines. The two other parameters are actually used when processing a word or a text.

a) Normalization means that the lexicon can match words without being case sensitive. Hence, this lexicon will recognize CLASS as a word.

b) The system has been implemented to recognize words in UTF8 encoding (actually the transducers are stored in Unicode). However, it is possible to tell the system how to take into account Latin encodings. For instance, you can provide the system with 5 as an encoding, which in this case refers to Latin 5, which is used to encode Cyrillic characters. The default value is Latin 1.

Vector

In the case of a vector as input to *add*, the structure will be a little different, the even positions in the vector will be the surface form, while the odd position will be the lemmas plus their features, again separated with a tab.

Map

For a map, the key will be the surface form, and the value the lemmas with their features. A map might actually prove a problem to store ambiguous words.

27.3 Processing strings

We have different ways of processing strings with a transducer.

lookup

lookup is used to detect if a word belongs to the transducer, and in this case it returns its lemma and its features. The transducer can return more than one solution. The recipient variable should be a vector in the case you want to retrieve all possible solutions.

Example:

t.lookup(“class”) returns: *class	+Singular+Noun*

You can constrain the processing of a string with edit distance threshold and actions.

t.lookup(“cliss”,1,a_change) returns: *class+Singular+Noun*

lookdown

lookdown is used to retrieve the correct suface form of a word using its lemma and its features.

Example:

t.lookdown(“class	+Plural+Noun”) returns: *classes*

parse

parse splits a string into tokens and returns for each token its lemma+features as a vector of all possibilities.

Example:

```
transducer t(_current+"english.tra");
string sentence="The lady drinks a glass of milk.";

vector v=t.parse(sentence);

printjln(v);
```

yields:

```
['The','The	+0+3+0+3+Prop+WordParticle+Sg+NOUN','the
 +0+3+0+3+Det+Def+SP+DET']
['lady','lady	+4+8+4+8+Noun+Sg+NOUN']
['drinks','drink	+9+15+9+15+Verb+Trans+Pres+3sg+VERB','drink
 +9+15+9+15+Noun+Pl+NOUN']
['a','a	+16+17+16+17+Det+Indef+Sg+DET']
```

```
['glass','glass   +18+23+18+23+Noun+Sg+NOUN','glass
  +18+23+18+23+Verb+Trans+Pres+Non3sg+VERB']
['of','of   +24+26+24+26+Prep+PREP']
['milk','milk      +27+31+27+31+Verb+Trans+Pres+Non3sg+VERB','milk
  +27+31+27+31+Noun+Sg+NOUN']
['.','.      +31+32+31+32+Punct+Sent+SENT']
```

N.B. *process* also returns the position of each word in the initial sentence.

27.4 Regular Expressions

The regular expressions processed by transducer are very limited:

1. %c: defines a character, c is a UTF8 character …
2. $.. : defines a string
3. u-u: defines an interval between two Unicode characters
4. [..]: defines a sequence of characters
5. {…}: defines a disjunction of strings
6. .+: structure should occur at least once.
7. (..): defines an optional structure
8. !n: inserts a features structure along its number in the feature vector (n>=1).

Examples:

```
transducer t;

//This expression recognizes Roman Numbers
t.compilergx("{DMCLXVI}+!1",["\t+Rom"]);

//This expression recognizes any kind of numbers including the decimal separator
and exponential expressions.
t.compilergx("({-+}){0-9}+!1(%.{0-9}+!2({eE}({-+}){0-
9}+!3))",["+Card","+Dec","+Exp+Dec"]);

//To recognize ordinal numbers
t.compilergx("{[1st][2nd][3rd]}!1",["+Ord"]);
t.compilergx("[3-9]([0-9]+)$th!1",["+Ord"]);
```

```
//we want to recognize any strings made of the Greek alphabet
t.compilergx("{α-ω0-9}+!1",["+Greek"]);

int i;
string s;
for (i in <945,970,1>)  s+=i.chr();


println(t.lookup("MMMDDD"));    //MMMDDD       +Rom
println(t.lookup("1234"));            //1234   +Card
println(t.lookup("1.234"));           //1.234  +Dec
println(t.lookup("1.234e-8"));       //1.234e-8         +Exp+Dec
println(t.lookup("1st"));               //1st      +Ord
println(t.lookup("2nd"));              //2nd      +Ord
println(t.lookup("3rd"));               //3rd      +Ord
println(t.lookup("4th"));               //4th      +Ord
println(t.lookup(s));                     //αβγδεζηθικλμνξοπρςστυφχψ +Greek
```

28 Type annotator

An annotator rule is a label associated with a body ending with ".":

(@|#|~)label <- e1,e2,e3.

Annotator Rules are divided into four groups:
- lexicons rules, which starts with a "@"
- annotation rules.
- Global rules, which start with a '#' and apply to existing labels
- Delete rules, which start with a '~' and remove existing annotations.

Rules are directly written as such in a Tamgu program. However, you need an "annotator" variable to access them.

28.1 annotator Methods

The type "annotator" exposes many methods to investigate what was extracted by the rules so far.

We have organized them is different sections.

Compiling

There is two ways to add rules in a program. Either, you type your rules directly into the code. In that case, there will be one single repository for all your rules. Or you can store them in a string and compile that string through an *annotator* variable.

1. **compile(string rules):** *the rules stored in rules are compiled into the annotator variable. Only this variable can access them.*

Such a string can also be passed to an annotator variable at declaration time:

```
annotator r(rulecode);
```

It will equivalent as to call the *compiling* method.

Rule Selection

It is possible to have different annotators in parallel each corresponding to a certain set of labels. For instance, you can associate to an annotator all rules, whose labels are lab1 and lab2. The annotator variable, which would have been associated with these labels, will only apply the rules whose head belongs to this selection.

2. **select(uvector labels)**: *select the rules, whose label is defined in labels*.
3. **clear()**: *clear the label selection.*
4. **selection()**: *return label selection.*

Lexicon

This method associates a general-purpose transducer lexicon to the annotator. A general-purpose lexicon is a lexicon of a given natural language, for instance: English, French or Korean. You can have more than one lexicon at a time. However, the fist lexicon, which is provided will be used to tokenize strings.

5. **lexicon(transducer t):** *Set the initial language dictionary, when available.*

Applying Rules

The two following methods are used to apply rules to either a string or a vector of strings.

6. **compilelexicons():** *this method pre-compiles the lexicons. If it is not called before a parse, then the first parse will compile them, which might introduce some delays.*
7. **spans(bool):**
   - *True return both the annotation classes and their offsets.*
   - *False, only the annotation classes.*

8. **parse(ustring txt,bool keeplex):** *apply rules to a ustring.*
9. **apply(uvector tokens,bool keeplex):** *apply rules to a vector of tokens.*
10. **apply(vector morphos,bool keeplex):** *apply rules to a vector of morphologically analyzed tokens. Each element is a vector containing at least two elements: [word, lemma, feat1,feat2..], where* word *is a asurface form,* lemma *its lemma form and* feat1, feat2.. *a list of features.*

    - **Example:** *the dogs*

      [["the", "the","det","definite"],["dogs","dog","noun","plural"]]

11. **apply(annotator,bool keeplex):** *apply rules to the structure that was computed by a previous annotator parse or apply. It allows for one single tokenization and different passes of rules.*

While parsing

While parsing, it is possible to have access to the internal structures, which have been extracted by the annotator so far. The following methods can be used in a callback function for instance to add more flesh to your analysis.

Label access

12. **checklabel(ustring label):** *Check if a label belongs to the annotated text.*
13. **labels():** *Return the list of all labels that were extracted.*

Tokens access

14. **words():** *Return the list of all words found in the text.*
15. **tokens():** *Return the sequence of tokens that were extracted from the text.*
16. **tokenize(ustring txt):** *Apply lexicons to txt and returns its tokenization. Also apply the associated function.*

Single token access

17. **checkword(ustring wrd):** *Check if a word belongs to the annotated text and returns the list of annotations.*
18. **word(ustring w):** *Return true is the word (or the lemma) has been detected in the text.*
19. **token():** *Return the current word or its index depending if the recipient variable is a string or an integer.*
20. **token(long idx):** *Return the token at the position idx itself...*

As dependencies

21. **dependencies(bool clear):** *dependencies stores in the knowledge base all the annotations as dependencies (see section 45 for more information on dependencies). Each token is transformed into a* synode *and each annotation is transformed into a dependency with the synodes as parameters. "clear" is optional, when it is true, the knowledge base is first emptied before receiving new entries. When an annotator is shared with another, its synodes are also shared, which means that new dependencies will share the same* synodes.

What is the difference between compiling and direct declaration?

Let's take an example.

a) First, we will declare the following rules, directly into the code:

```
a_food ← #food, {#food, with, [from, (the), #place+]}*.
a_service ← #service+.
a_ambience ← #ambience+.
a_resto ← #resto+.
a_price ← #price+.

annotator r;
annotator rbis;
```

In the above case, *r* and *rbis* will share the same set of rules. There will be one single repository for all these rules in which to be stored.

If we want these annotators to access specific rules, then you need to use the method: *select.*

```
r.select(["a_place","a_food","a_service","a_ambience","a_resto","a_price"]);
rbis.select([""a_negative"]);
```

In our example, *r* and *rbis* will only apply rules whose head category falls into their selection vector. *rbis* for instance will only apply rules that yield a "*a_negative*" label.

b) The second way to declare these rules is to use *compiling.* In that case, we need of course string variables in which to store the different rules.

```
string rule1=@"
a_place ← >{in, from, at}<, "%C%a+", {"%C%a+", ["-","%C%a+"],
of,from,the,with}+.

a_food ← #food, {#food, with, [from, (the), #place+]}*.
a_service ← #service+.
a_ambience ← #ambience+.
a_resto ← #resto+.
a_price ← #price+.
"@;

string rule2=@"
a_negative <- #negative+.
"@;

//Simply words associated with a "lexicon label"

//All these rules will be access through an annotator
annotator r1(rule1);
annotator r2;
r2.compling(rule2); //equivalent to above
```

In the above example, r1 will comprise the rules in *rule1*, and r2 the rules in *rule2*. In this case, the two *annotators* will not share a same rule repository, they will have their own.

28.2 Callback Function

You can associate a callback function to an annotator object.

Important: This callback function can only be used if lexicons have been provided.

An *annotator* callback function is called *after the parsing* took place, but before the rules to apply to the texts.

This callback function has the following signature:

```
//Simply words associated with a "lexicon label"
function catching(vector v, annotator a) {
    v=pos.tagger(v);
    return(v);
}


//All these rules will be access through an annotator
annotator r with catching;
```

The vector *v* is a list of vectors, where each sub-vector contains:

[word, *lemma1, features1, lemma2, features2,..,*leftoffset,rightoffset]

“word” is the token that was extracted from the text, while *lemma1, features1,etc.* are potential readings of this token. *Leftoffset* and *rightoffset* are the position in characters of that word within the text.

The vector that is returned is expected to have the same structure.

Example:

```
['about','about','+Adv+notly+ADV','about','+Prep+PREP','39','44'],
```

This function can be used to apply *tagging*, for instance, to the analysis beforehand.

28.3 Syntax

A rule in Tamgu is a regular expression, which combines tokens and semantic categories as defined in the lexicons. A rule returns a label, which is associated with the span of the token sequence that was recognized.

The body is written after the "<-". However the character "←" can also be used.

```
goodfood <- #food,?+,delicious.
```

For instance, this rule says that if something is a *food* and is followed by “was delicious” then the label **goodfood** is produced for this sequence of words. "?+" means a sequence of at least one token.

Operators

The operators are the following:

- {t1,t2,t3..}:	The token must match one of the elements in the list of token.
- ?:	Any token.
- #label:	A semantic label either produced with a rule or the lexicon
- #{l1,l2..}	If the label is one of l1,l2… (disjunction)
- #[l1,l2..]	The label should l1 and l2 and so on… (conjunction)
- *,+ (:c):	Kleene operators, which can be bounded with a counter…
- ~:	negation (only of atomic elements)
- (..):	Optional sequence of tokens
- [..]:	Sequence of tokens
- \>..<:	The tokens in this sequence will be skipped in the final annotation.
- \<call p1 p2\>:	A call to a function that returns true or false…
- token:	A simple token, which must match our current word from the text
- ‘rgx’:	A regular expression based on posix regular expression
- “metas”:	A Tamgu specific regular expression based on the following meta-characters (see below).

%p:	A punctuation
%d:	A digit
%C:	An uppercase character
%c:	A lowercase character
%a:	An alphabetical character
%e	An emoji character

%H:	A Hangul character
%s:	A space character
%r:	A carriage return
%S:	both space and carriage return
%x:	stands for a hexadecimal digit (0-9a-fA-F)
%X:	escape the character X, where X is any character…
~X:	negation
\x	escape character
\ddd	character code across 3 digits exactly
\xFFFF	character code across 4 hexas exactly

28.4 Tamgu Regular Expression vs. Posix Regular expressions

Tamgu regular expressions provide a very simple schema to handle character expressions. A Tamgu regular expression can handle more than one word at a time.

Tamgu exposes a list of meta-characters (see above for a list of them) that can be combined with characters and Kleene operators. A Tamgu regular expression is always encapsulated with "" (double quotes).

Example

"%p+" : a string composed of only punctuations.
"%H+" : a string composed of only Hangul characters.
"test(s)": a string with an optional 's'.
"Programming Language": a multiword expressions.

Note that the first example: "%p+" is equivalent to %p+. You can omit the quotes in this case. The same applies to %H+. However, if you have a combination of characters and multiple meta characters, you must use the double quotes.

Posix Regular expressions

Posix regular expressions are also available, but they cannot be used for tokenisation process.

These expressions are embedded within '' (single quotes).

Example:

'(\d{1,3}):(\d{1,3}):(\d{1,3}):(\d{1,3})' this expression can recognized: 1:22:33:444

28.5 Lexicon rules

A lexicon rule is used to describe domain vocabulary. The body of a lexical rule is one element only. However, you can declare multiword expressions as Tamgu regular expressions.

Important: Only Tamgu regular expressions can be used for multiword expressions.

However, if you want your expression to be part of the tokenisation process (i.e. so that mwe can be recognized as one single token), only Tamgu regular expressions can be used.

Examples

```
//Simple words associated with a "lexicon label"
@positive ← great.
@positive ← better.

//Another category
@food ← sushi.

//This is a regular expression, the "s" is optional
//".." is a TAMGU regular expression
@food ← "food(s)".
```

28.6 Annotation Rules

An annotation rule is a label associated with a body. The body can integrate different elements such as optional elements, disjunction of elements, regular expressions, skipped words etc. It returns a list of labels with their spans in the original text as offsets.

Output

When a rule applies to a sequence or tokens, or to a token, each **token** is then associated with the *label*.

In other words, we can see *labels* as classes. When a rule applies, the tokens from the initial sequence that are spanned by this rule are distributed along the *label* or *class* of this rule.

```
//Initial sequence is: the chicken was delicious. The chicken was
very soft.

//Rules:

@food <- chicken.

good <- #food, ?, delicious.

//aspect can now benefit from the application of the above rule
//note that we restrict the number of tokens between good and soft
//to a maximum of three...
aspect <- #good, ?+:3, soft.

// When the rule "good" applies, the token "chicken"
//receives the "good" label and can then be identified
//through this new label, in subsequent rules.
```

Example

```
//// Optional elements, #food matches a rule label or a lexicon label
food ← (#positive), #food+.

// We can skip words: >..<
// We can also specify that a word can match different labels.

service ← #personnel, >?*:5<, #[positive,service].
```

Function Callbacks

A rule can call a function, which will investigate the current token, through more complex kinds of matching. For instance, a function might compare the current token against specific lexicons or against word embedding.

The function must have at least two parameters. The first one is the current token, the second one is the current annotator variable. However, it is possible to provide more variables if necessary.

```
function distance(string currenttoken, annotator a) {
    if (currentoken.editdistance('is')<=2)
        return(true);
    return false;
}


label1 <- this, <distance>, a, thing.

// This rule will yield label1 for the following utterances:

// this is a thing
// this ist a thing
// this sis a thing
```

We can also implement a more general version of this function:

```
//We replace the "is", which was hard-coded with a variable
function distance(string currenttoken, annotator a, string s) {
    if (currentoken.editdistance(s)<=2)
        return(true);
    return false;
}

//we call the function with one more parameter...
label1 <- this, <distance "is">, a, thing.
```

This function can implement any kind of behaviour. For instance, the token could be compared against word embedding or against a cosine distance between two words.

Global Rules

Global rules that start with a "#" or a "~" are similar to annotation rules, however, they are not applied along the text, but once the text has been fully processed to handle annotations in order to add new ones or remove existing ones.

```
//Our function called from a global rule, one single parameter
function test(annotator a) {
    if (a.word("bread"))
        return(true);
    return(false);
}


#label ← #lab1, #lab2, <test>.
```

For instance, the above rule will create the annotation *label*, if *lab1* and *lab2* have been created and *call* returns *true*.

Note that *call* in this case has only one parameter, since the rule applies independently from the text.

A Code example

```
//The lexicon...

//Simply words associated with a "lexicon label"
@positive ← great.
@positive ← better.
@positive ← good.
@positive ← accomodating.

//We compare the words with their lemma
@negative ← $overrate.
@negative ← $disappoint.

//Another category
@food ← sushi.

//This is a regular expression, the "s" is optional
//".." is a TAMGU regular expression
//'..' is a posix regular expression
@food ← "food(s)".

//These are multiword expressions, which will be combine with the
lexicon
//For a multiword tokenization
@food ← "lemon chicken".
@food ← "honey walnut prawn(s)".

@service ← accomodating.

@personnel ← staff.
@personnel ← personnel.
```

```
@personnel ← $waiter.

//Rules...
// Optional elements, #food matches a rule label or a lexicon label
food ← (#positive), #food+.

// We can skip words: >..<
// We can also specify that a word can match different labels.
service ← #personnel, >?*:5<, #[positive,service].

//A disjunction is expressed with {...}
//%p is a punctuation.
positive ← "can%pt", say, enough, {of,about}.
positive ← no, $complaint.
service ← $tend, >?+:5<, my, $need.

negnot ← "not", #positive.
neg ← #negative.

//All these rules will be access through an annotator
annotator r;
//We load our english dictionary
transducer lex(_current+"english.tra");
//Which we associate with our annotator...
r.lexicon(lex);

ustring u=@"
No Comparison...
I can't say enough about this place.
It has great sushi and even better service.
The waiters were extremely accomodating and tended to my every
need.
I've been to this restaurant over a dozen times with no complaints to
date.

Overrated and not good.
– I was highly disappointed in the food at Pagoda.
The lemon chicken tasted like sticky sweet donuts and the honey
walnut prawns, the few they actually give you.....were not good.
The prices are outrageous, especially since the food was actually
less satisfying than most neighborhood Chinese establishments.
Nice ambience, but highly overrated place.
I will not go back.
The waiters were extremely accomodating and tended to my taste
and to my need.

"@;
```

```
		//We apply our rules to the text above...
		vector res=r.parse(u,true);
		printjln(res);
		printjln();

		//We store our annotations as dependencies...
		r.dependencies();
		vector vdeps = predicatedump();
		printjln(vdeps);

		//We can now use dependency rules to handle these annotations
```
The output is the following:

```
['positive',[20,25],[26,29],[30,36],[37,42]]
['food',[62,67],[68,73]]
['positive',[221,223],[224,234]]
['neg',[245,254]]
['negnot',[259,262],[263,267]]
['neg',[284,296]]
['food',[304,308]]
['food',[324,337]]
['food',[378,397]]
['negnot',[439,442],[443,447]]
['food',[497,501]]
['neg',[604,613]]
['service',[685,691],[711,713],[714,718]]

neg({"trans":"+","123sp":"+","vpap":"+","adj":"+","pastboth":"+","idx":"48","adj
pap":"+","lemma":"overrate","verb":"+","lemma1":"overrated","vpast":"+","sur
face":"Overrated"})
food({"surface":"honey walnut prawns","food":"+","lemma":"honey walnut
prawns","idx":"73"})
food({"noun":"+","surface":"food","sg":"+","lemma":"food","idx":"95"})
```

etc…

Note : We have displayed only a sample of the actual content of *predicatedump.*

29 Type grammar

grammar is a type that is designed to provide coders with a powerful way to describe complex string structures.
For instance, if you need to detect specific sub-strings in a text, which involves digits, upper case letters, or punctuations in a strict order, then *grammar* will definitely help you.

29.1 Methods

There are only two functions that are exposed by this type:

1. **apply(string|vector):** *you can apply a grammar to a text, which will be transformed into a vector of characters, or to a vector of tokens.*

2. **load(rule, int skipbanks):** *you can either load rules as a string or as a vector of rules. You can also load rules when building the grammar object itself. skipblanks is optional, it can have the following values:*

   0*: then all characters should be taken into account in the grammar. This is the default value, when skipblanks is omitted.*

   1: *white spaces and tabs are automatically skipped, before applying a target to a sub-strings. Trailing characters at the end of the strings are also skipped.*

   2: *all spaces including carriage returns are skipped.*

Note: the "in" operator can also be used with a grammar. It is then used as a way to detect if a string is compatible with the grammar.

29.2 Rules

Rules are implemented either as a single text (which is the easiest way) or as a vector of strings, each string is then a rule.

Rule format

The format of a rule is the following:

head := (~) element [,;] element .

where element is:

a string =	between quotes “a” or ‘a’	
?	=	any character
%a	=	any alphabetic character
%c	=	any *l*ower case character
%C	=	any *U*pper case character
%d	=	a digit
%e	=	an emoji character
%H	=	a Hangul character
%r	=	a carriage return
%s	=	a space character
%S	=	a separator character (space or carriage return)
%p	=	a punctuation
%?	=	the “?” character
%%	=	the “%” character
0,1,2..9	=	any digit, which is actually a character code
$string	=	a string of any length (same as “string”)
head	=	the head of another rule

- Negation: All these elements can be negated with “~” except heads.
- Disjunction: You use the “;” when you need a disjunction between two elements, a “,” otherwise.
- Kleene star: You can use “+” or “*” to loop for each of these elements.
    - Longest match: If you use “++” or “**”, then the loop will consume the string up to the most reachable element.
- Optional: You can use “(element)” for optional characters or heads.
- All rules should end with a “.”.
- When a head name starts with a “_”, then the string is extracted, but its label is not stored.

Specific cases:

?_	=	any character, but not stored
%a_	=	any alphabetic character, but not stored
%c_	=	any *l*ower case character, but not stored
%C_	=	any *U*pper case character, but not stored
%d_	=	a digit, but not stored
%e_	=	an emoji, but not stored
%H_	=	a Hangul character, but not stored
%r_	=	a carriage return, but not stored

%s_	=	a space character, but not stored
%S_	=	a separator character, but not stored
%p_	=	a punctuation, but not stored
label_	=	a call to a rule, without storage

The adjunction of a "_" at the end of these options allows for a recognition of a character or a group of characters, which is, however, not stored in the final result.

Example

```
//This grammar recognizes a word or a number, only for one string...
string r=@"

bloc  := word;number.
word := %a+.
number := %d+.

"@;

//we load our grammar
grammar g(r);

//we apply it to the string the
map m=g.apply("the"); //it returns: {'bloc':[{'word':['the']}]}

m=g.apply("123"); //it returns: {'bloc':[{'number':['123']}]}
```

However, if we apply this grammar to: "Test 123", it will fail. We need to add to this grammar two things:

a) First, it should take into account spaces
b) Second, it should loop to recognize every token in the string

```
string r=@"

base := bloc+.
bloc  := word;number;%s.
word := %a+.
```

```
number := %d+.

"@;
```

We have added a new disjunction with %s to take into account spaces. Then we have added a “base” rule that loops on bloc.

If we apply our grammar to: “Test 123”, then the system will return:

{'base':[{'bloc':[{'word':['Test']}]},{'bloc':[' ']},{'bloc':[{'number':['123']}]}]}

N.B. There is another way to skip the blanks, you can declare your grammar with: *grammar g(r,1)*; In that case, the call to “%s” is useless.

However, the structure might be a bit difficult to assess. We can then use the “_” operator to remove from this output the unnecessary information, such as “bloc”.

```
string r=@"

base := bloc+.
bloc  := word;number;%s.
word := %a+.
number := %d+.

"@;
```

In this grammar, _*bloc* is now a hidden head and if we apply this grammar to our input, the result is:

{'base':[{'word':['Test']},' ',{'number':['123']}]}

We could also decide to enrich our number structure with a more refined set of information with number words such as million, billion or thousand. In this case, we will put number as the first element of the _bloc structure to detect these specific strings.

```
string r=@"
base := _bloc+.
_bloc  := number;word;%s.
```

```
word := %a+.
number := %d+;$billion;$millions;$thousand.
"@;
```

If we apply this grammar to: “Test millions of cows”, we obtain:

{'base':[{'word':['Test']},' ',{'number':['millions']},' ',{'word':['of']},' ',{'word':['cows']}]}

If we want to recognize more complex structure, such as a code, which would start with an uppercase and be followed by digits, then we could implement the following grammar:

```
string r=@"

base := _bloc+.
_bloc  := code;word;number;%s.
word := %a+.
number := %d+.
code := %C,%d+,%c.
"@;
```

If we apply this grammar to: “Test 123 T234e”, we get:

{'base':[{'word':['Test']},' ',{'number':['123']},' ',{'code':['T234e']}]}

29.3 Sub-grammars

Sub-grammars are introduced within [...]. In these brackets, it is possible to define a disjunction of character regular expression strings. These expressions are especially useful when you apply a grammar to a vector of strings, in this case, string can be matched at the character level against the expression itself. Each expression should be separated from the following with a “|”.
You cannot call a rule from within brackets, therefore a string such as *dog* will be equivalent to *$dog*.

Example:

```
string dico=@"
test := %a, wrd,%a.
```

```
wrd := [%C,("-"),%c+|test|be|dog|cat].
"@;

grammar g(dico);
ustring s="The C-at drinks";
uvector v=s.tokenize();
vector res=g.apply(v);
println(res);
```

Vector vs. Map

If we replace the recipient variable with a vector, then the output is rather different. The head rule name is inserted into the final structure as the first element. Hence, if we apply the above grammar to the same string, but with a vector as output, we obtain:

['base',['word','Test'],' ',['number','123'],' ',['code','T234e']]

29.4 Input is a string or a vector

If the input is a string, then each detected character is appended to the output string. However, if the input is a vector of characters, we keep the output result as a vector of characters.

Example

```
//This grammar recognizes a word or a number
string r=@"

base := _bloc+.
_bloc  := code;word;number;%s.
word := %a+.
number := %d+.
code := %C,%d+,%c.

"@;

//we load our grammar
grammar g(r);
```

```
//we split a string into a character vector
string s="Test 123 T234e";
svector vs=s.split("");

//we apply the grammar to the character vector
vector  v=g.apply(vs);
println(v);
```

The output in this case is:

```
['base',['word','T','e','s','t'],' ',['number','1','2','3'],' ',['code','T','2','3','4','e']]
```

29.5 Function

It is also possible to associate a function with a grammar. The signature of the function is the following:

function grammarcall(string head, self structure,int pos).

This function is called for each new structure computed for a given head. If this function returns *false*, then the analysis of that rule fails. *pos* is the last position in the string up to which parsing did take place.

Example

```
//This grammar recognizes a word or a number
string r=@"

base := _bloc+.
_bloc  := code;word;number;%s.
word := %a+.
number := %d+.
code := %C,%d+,%c.

"@;

//This function is called for each new rule that succeeds
function callgrm(string head,self v,int ps) {
    println(head,v,ps);
    return(true);
}

//we load our grammar
grammar g(r) with callgrm;

//we split a string into a character vector
string s="Test 123 T234e";
//we apply the grammar to the character vector
map m=g.apply(s);
println(m);
```

Result:

```
word ['Test']
_bloc [{'word':['Test']}]
_bloc [' ']
number ['123']
_bloc [{'number':['123']}]
_bloc [' ']
code ['T234e']
_bloc [{'code':['T234e']}]

{'base':[{'word':['Test']},' ',{'number':['123']},' ',{'code':['T234e']}]}
```

Modification of the structure

You can also modify the structure in this function, but you should be careful of your modifications…

```
function callgrm(string head,self v,int ps) {
    //If the head is a word, we modify the inner string
    if (head=="word") {
        println(head,v);
        v[0]+="_aword";
    }
    return(true);
}
```

Then in this case, the output is:

word ['Test']

{'base':[{'word':['Test_aword']},' ',{'number':['123']},' ',{'code':['T234e']}]}

From within a rule

A function can also be called from within a rule. The signature is the following:

function rulecall(self structure,int pos).

```
//This function is called from within the code rule…
//If it returns false, then the code rule fails.
function callcode(self v,int ps) {
    println(head,v);
    return(true);
}

//This grammar recognizes a word or a number
string r=@"
base := _bloc+.
_bloc  := code;word;number;%s.
word := %a+.
number := %d+.
code := %C,%d+,%c,callcode.
"@;

//we load our grammar
grammar g(r);

//we split a string into a character vector
string s="Test 123 T234e";
//we apply the grammar to the character vector
map m=g.apply(s);
println(m);
```

Example: parsing HTML

//evaluate is a basic method to replace every HTML entity with its UTF8 counterpart.

```
function evalue(self s,int p) {
    s[1]=s[1].evaluate();
    return(true);
}

//This is our HTML grammar
//We do not keep space characters between tag, hence: %s_ in object
string htmlgrm=@"

html := _object+.
_object := tag;%s_;text.
tag := "<",?+,">".
text := _characters,evalue.
_characters := ~"<"+.

"@;

//We compile our grammar
grammar ghtml(htmlgrm);

//which we can apply to an html text
vector rgram=ghtml.apply(html_text);
```

30 Type iterator, riterator

These iterators are used to i**terate on any objects of type:** *string, vector, map, rule.*

riterator is the reverse iterator, which is used to iterate from the end of the collection.

30.1 Methods

1. **begin()***: initialize the iterator with the beginning of the collection*
2. **end()***: return true when the end of the collection is reached*
3. **key()***: return the key of the current element*
4. **next()***: next element in the collection*
5. **value()***: return the value of the current element*

Initialization

An iterator is initialized through a simple affectation.

Example

```
vector v=[1,2,3,4,5];
iterator it=v;
for (it.begin();it.nend();it.next())
   print(it.value(),",");
```

Run
1,2,3,4,5,

31 Type date

This type is used to handle dates.

31.1 Methods

1. **date()**: *return the date as a string*
2. **day()**: *return the day as an integer*
3. **format(string f)**: *return the format as a string. The format string uses a combination of options. See below for an explanation.*
4. **hour()**: *return the hour as an integer*
5. **min()**: *return the min as an integer*
6. **month()**: *return the month as an integer*
7. **sec()**: *return the sec as an integer*
8. **setdate(year,month,day,hour,min,sec)**: *set a time variable*
9. **year()**: *return the year as an integer*
10. **yearday()**: *return the year day as an integer between 0-365*
11. **weekday()**: *return the week day as an integer between 0-6. 0 is Sunday.*

Operators

+,-: dates can be added or subtracted

As a string

return the date as a string

As an integer or a float

return the number of seconds elapsed since 00:00 hours, Jan 1, 1970 UTC

Format

%a: The abbreviated weekday name according to the current locale.

%A: The full weekday name according to the current locale.

%b: The abbreviated month name according to the current locale.

%B: The full month name according to the current locale.

%c: The preferred date and time representation for the current locale.

%C: The century number (year/100) as a 2-digit integer. (SU)

%d: The day of the month as a decimal number (range 01 to 31).

%D: Equivalent to **%m/%d/%y**. (Yecch-for Americans only. Americans should note that in other countries **%d/%m/%y** is rather common. This means that in international context this format is ambiguous and should not be used.) (SU)

%e: Like **%d**, the day of the month as a decimal number, but a leading zero is replaced by a space. (SU)

%E: Modifier: use alternative format, see below. (SU)

%F: Equivalent to **%Y-%m-%d** (the ISO 8601 date format). (C99)

%G: The ISO 8601 week-based year (see NOTES) with century as a decimal number. The 4-digit year corresponding to the ISO week number (see **%V**). This has the same format and value as **%Y**, except that if the ISO week number belongs to the previous or next year, that year is used instead. (TZ)

%g: Like **%G**, but without century, that is, with a 2-digit year (00-99). (TZ)

%h: Equivalent to **%b**. (SU)

%H: The hour as a decimal number using a 24-hour clock (range 00 to 23).

%I: The hour as a decimal number using a 12-hour clock (range 01 to 12).

%j: The day of the year as a decimal number (range 001 to 366).

%k: The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are preceded by a blank. (See also **%H**.) (TZ)

%l: The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are preceded by a blank. (See also **%I**.) (TZ)

%m: The month as a decimal number (range 01 to 12).

%M: The minute as a decimal number (range 00 to 59).

%n: A newline character. (SU)

%O: Modifier: use alternative format, see below. (SU)

%p: Either "AM" or "PM" according to the given time value, or the corresponding strings for the current locale. Noon is treated as "PM" and midnight as "AM".

%P: Like **%p** but in lowercase: "am" or "pm" or a corresponding string for the current locale. (GNU)

%r: The time in a.m. or p.m. notation. In the POSIX locale this is equivalent to **%I:%M:%S %p**. (SU)

%R: The time in 24-hour notation (**%H:%M**). (SU) For a version including the seconds, see **%T** below.

%s: The number of seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC). (TZ)

%S: The second as a decimal number (range 00 to 60). (The range is up to 60 to allow for occasional leap seconds.)

%t : A tab character. (SU)

%T: The time in 24-hour notation (**%H:%M:%S**). (SU)

%u: The day of the week as a decimal, range 1 to 7, Monday being 1. See also **%w**. (SU)

%U: The week number of the current year as a decimal number, range 00 to 53, starting with the first Sunday as the first day of week 01. See also **%V** and **%W**.

%V: The ISO 8601 week number (see NOTES) of the current year as a decimal number, range 01 to 53, where week 1 is the first week that has at least 4 days in the new year. See also **%U** and **%W**. (SU)

%w: The day of the week as a decimal, range 0 to 6, Sunday being 0. See also **%u**.

%W: The week number of the current year as a decimal number, range 00 to 53, starting with the first Monday as the first day of week 01.

%x: The preferred date representation for the current locale without the time.

%X: The preferred time representation for the current locale without the date.

%y: The year as a decimal number without a century (range 00 to 99).

%Y: The year as a decimal number including the century.

%z: The *+hhmm* or *-hhmm* numeric timezone (that is, the hour and minute offset from UTC). (SU)

%Z: The timezone or name or abbreviation.

%+: The date and time in **date** format.

%%: A literal '%' character.

Example 1

```
date d;
println(d.format("%Y%m%d")); //display date for 2015/12/25 as 20151225
```

Example 2

```
date mytime;
print(mytime);    // display: 2010/07/08 15:19:22
```

32 Type time

This type is used to compute timeframes or duration.

32.1 Methods

1. **reset ()**: *reinitialize a time variable*

Operators

+,-: time can be added or subtracted

As a string

return the time in ms

As an integer or a float

return the time in ms

Example

```
time mytime;
print(mytime);
```

33 Type chrono

This type is quite similar to *time* but is implemented on the basis of *std::chrono::high_resolution_clock*. It exposes the same method *reset* however it allows for a more refined time computing.

When you create a *chrono* object, you need to specify how the duration should be handled, with the following flags:

- c_second
- c_millisecond
- c_microsecond
- c_nanosecond

The default value is: *c_second*.

Example:

```
ring r;

chrono c1(c_nanosecond);
for (int i in <100000>)
   r.pushlast(i);
chrono c2(c_nanosecond);

float d;
d = c2-c1; //in nanoseconds
println(d);
```

34 Type file, wfile

This type is used to manage a file in input and output. The type “wfile” is used to handle UTF16 (UCS-2 more precisely) files.

34.1 Methods

1. **eof()**: *return true when the end of file is reached*
2. **file f(string filename, string mode_read)**: *open a file according to moderead. If the file is in read mode, then “moderead” is optional. The possible values for moderead are:*
    a. “a”: append
    b. “r”: read
    c. “w”: write
    d. “w+”: append
3. **find(string s,bool caseinsensitive)**: *return all positions in the file of the string s.*
4. **get()**: *read one character from the file*
5. **getsignature()**: *return whether the file contains a signature*
6. **openappend(string filename)**: *open a file in append mode*
7. **openread(string filename)**: *open a file in read mode*
8. **openwrite(string filename)**: *open a file in write mode*
9. **read()**: *read the whole file into a variable, which can be:*

a. **string**: *the whole document is store in one string*

b. **svector**: *the document is split into string along carriage returns, which are each stored into the container.*

c. **bvector**: *the document is stored byte by byte into the container.*

d. **ivector**: *the document is stored byte by byte into the container.*

10. **read(int nb)**: *like read, but extracts only "nb" characters or bytes from the file.*

11. **readln()**: *read a line from a file*

12. **seek(int p)**: *position the file cursor at p*

13. **setsignature(bool s)**:*set the UTF8 or UTF16 signature (accordingly)*

14. **tell()**: *return the position of the file cursor*

15. **unget()**: *return one character to the stream*

16. **unget(nb)**: *return nb character to the stream*

17. **scan(string grammar, string separator):** *read a file with a Tangu Regular Expression. Returns a vector of substrings. The separator is a character that separates a field from another. It is a space by default. See below for an example.*

18. **write(string s1,string s2,…)**: *write strings in the file*

19. **writelen(string s1,string s2,…)**: *write strings in the file, separating each string with a space, and adding a carriage return at the end of the line.*

20. **writebin(int s1,int s2,…)**: *write bytes in the file. If the value is a container, then write the list of bytes out of that container.*

signature

UTF-8 and UTF-16 files might have a signature at the beginning, which consists of three octets to define a UTF-8 file or two octets in the case of a UTF-16 file.

- If you use the type *"file"*, then in order to read the signature out, you must set the signature beforehand. This type can only be used to read UTF-8 or binary files.

- In the case of *"wfile"*, the signature is automatically set when the signature is found at the beginning of the file. You can only read UTF-16 (UCS-2) files with this type.

Operator

x in file: if x is a string, then it receives a line from the file, if it is a vector, it pushes the line on the top of it. If x is an integer or a float, it gets only one character from the stream.

Example

```
file f;
f.openread(path);
string s;
svector words;
string w;
for (s in f) {//Using the in operator
    s=s.trim();
    words=s.split(" ");
    for (w in words)
        print("word:",w,endl);
}
f.close();
```

34.2 Standard input: stdin

Tamgu provides the variable *stdin* to handle the standard input. This variable can be quite useful to handle data coming from a piped file for instance.

Example

```
string s;
int i=1;
for (s in stdin) {
    println(i,s);
    i++;
}
```

If you store these lines in a small file say: stdin.Tamgu, then the content of the piped strings will be displayed with for each line a specific number:

echo “The lady is happy” | Tamgu stdin.Tamgu.

Example (scan)

The text contains lines such as:

```
456 -0x1.16bca4f9165dep-3 0x1.0d0e44bfeec9p-3
```

```
file f(_current+"tst.txt");
//we define a macro to read a complex hexadecimal string
grammar_macros("X","({%+-})0x%x+(%.%x+)(p({%+-})%d+)");

//We read the two first fields... The space is our default separator
uvector v=f.scan("%d+ %X ");
println(v);                          //['456','-0x1.16bca4f9165dep-3']

//The next field...
v=f.scan("%X ");
println(v);                          //['0x1.0d0e44bfeec9p-3']

f.close();
```

35 Type call

This object is used to keep tracks of functions, which can then be executed. The call is done using the variable name as a *function.*

Example

```
function display(int e)  {
   print("DISPLAY:",e,"\n");
   e+=10;
   return(e);
}

call myfunc;
myfunc=display;
int i=myfunc(100);                  // display: DISPLAY:TEST
print("I=",i,"\n");                 //display: I=110
```

36 Specific instructions

Tamgu provides all the necessary operations to handle all sorts of algorithms: *if, else, elif, switch, for, while.*

Boolean operators: and (&&), or (||), xor

These Boolean operators combine different elements with a Boolean value together. The *and* and the *or* operators can also be written as && or ||.

- and: succeeds if all elements are true
- or: succeeds if at least one element is true
- xor: succeeds if at least one element is true but fails if they are all true.

if—elif—else

if (booleanexpression) {}

elif (booleanexpression) {}

…

else {}

ifnot

ifnot is used to chain different instructions and returns the value of the first instruction that returns anything but *false, null* or *empty.*

Call1() ifnot call2() ifnot call3()…

Note, that in the case of a complex sequence of operations, you must isolate the whole sequence of "ifnot" between parentheses.

Example:

```
map m1={"a":1,"b":2,"c":3};
map m2={"A":4,"B":5,"C":6};
map m3={"aa":7,"bb":8,"cc":9};

int i= (m1["A"] ifnot m2["B"] ifnot m3["bb"])+24;
println(i); //tamgu returns: 29
```

```
function calling(string s) {
    return(m1[s] ifnot m2[s] ifnot m3[s]);
}

println(calling('bb')); //tamgu returns: 8
```

switch (expression) (with function) {…}

The *switch* enables to list a series of tests for one single object:

```
switch(expression) {
   v1 : {…
  }
   v2 : {…
  }
  default: {…    //default is a predefined keyword
     }
}
```

v1,v2,..vn can be either a string or an integer or a float. The expression is evaluated once and compared with v1, v2, vn…

It is also possible to replace the simple comparison between the elements with a call to a function, which should return *true* or *false*.

```
//we test wether one value is larger than the other
function tst(int i,int j) {
  if (j>=i)
      return(true);
  return(false);
}
int s=10;
//We test through test
switch (s) with tst {
  1:  println("1");
  2:  println("2");
  20:  println("20"); //This will be the selected occurrence
}
```

36.1 “for” operators.

There are different flavours of “for” in Tamgu. Here is a presentation of them all.

for (expression;boolean;next) {…}

This *for* is composed of three parts, an initialisation, a Boolean expression and a continuation part.

You can use continue or break to either go to the next element or to break in the middle of a loop.

Example

```
for (i=0;i<10;i+=1) print("I=",i,"\n");
```

Multiple initializations and increments

Expressions, both in the initialization part and in the increment part, can contain more than one element. In that specific case, these elements should be separated by a comma.

Example

```
int i,j;

//Multiple initializations and multiple increments.
for (i=10,j=100;i>5;i--,j++)
  println(i,j);
```

for (var in container) {…}

This is a very specific sort of *for,* which is used to loop in a container, a string or a file.

Example

```
//we loop in a file
file f('myfile.txt',"r");
string s;

for (s in f)
   println(s);
```

```
//we loop in a vector of ints...
vector v=[1,2,3,4,5,6];
int i;

for (i in v)
    println(I);
```

for (i in <start,end,increment>): Fast loop

This loop is equivalent to: for (i=start;i<end;i+=increment)…

Actually, the loop can also be equivalent to: *for (i=start;i>end;i-=increment)* if the increment is negative.

The reason for this loop is that it is implemented as a C++ loop, and is about 30% to 50% faster than its equivalent. Each of the values in the range can be instantiated through variables; however, once the loop has started no element can be modified, including the variable which receives the different values.

Example:

```
int i,j=1;
int v;

time t1;

//Looping to 100000, with an increment of 1.
for (i in <0,100000,j>)
    v=i;

time t2;
float diff=t2-t1;
println("Elapsed time for fast ‘for’:",diff);

time t3;
for (i=0;i<100000;i+=j)
    v=i;

time t4;
diff=t4-t3;
println("Elapsed time for regular ‘for’",diff);
```

Local declarations

You can also declare variables into a "for" statement, which are only to the "for" code.

Example:

```
for (int i in <0,100000,j>) println(i);
for (int i=0;i<10;i++) println(i);
```

Default arguments

It is also possible to omit some of the arguments in the "for" for more succinct code writing.

- for (k in <100>)… means that initial value is 0 and increment is 1
- for (k in <1,100>)… means that increment is 1 by default

36.2 while (boolean) {…}

while is composed of a single Boolean expression.

while (boolean) {…}

You can use continue or break to either go to the next element or to break in the middle of a loop.

Example

```
int i=10;
while (i>0) {
   print("I=",i,"\n");
   i-=1;
}
```

36.3 do {…} while (boolean);

This expression is similar to *while*, however, the first iteration is done before the Boolean test.

Example

```
int i=10;
do {
   print("I=",i,"\n");
   i-=1;
}
while (i>0);
```

36.4 Evaluation: eval(string code);

This function can evaluate and run some Tamgu *code* on the fly. The result of the evaluation is returned according to what was evaluated.

36.5 JSon Evaluation: evaljson(string code)

This function takes a JSON string as input and compile either into a vector or a map according to the inner structure.

36.6 print, println, printerr,printlnerr

These instructions are used to display results on the current display port. The “err” versions display the results on the standard error output. The “ln” version add two features to the output, for the values separated with a “,”, an additional space is added. Second, a carriage return is added at the end of the line.

36.7 printj, printjln, printjerr,printjlnerr

These versions are quite different from the previous one. The “j” stands for a *join*. These instructions are used to display container values, which are “joined” beforehand. They accept either two, three or four arguments. The first parameter should be a container and the second one a separator string. If the container is a map, then a key separator can also be supplied as third parameter. A fourth numerical parameter (or a third for vectors and strings) can be provided that will add a carriage return every *n* values.

If only the container is supplied, then the default separator is the carriage return.

- printj(string); //sep is CR
- printj(string, “sep”);
- printj(string, “sep”, counter);
- printj(vector); //sep is CR
- printj(vector, “sep”);
- printj(vector, “sep”, counter);
- printj(map); //sep is CR, value-sep is “:”
- printj(map, “sep”);
- printj(map, “sep”, “value-sep”);
- printj(map, “sep”, “value-sep”, counter);

Example

```
ivector v=[1..10];
printj(v,"-");



Result is: 1-2-3-4-5-6-7-8-9-10


map m={1:2,2:3,4:5,6:7};
printjln(m,",","-");
```

Result is: 1-2,2-3,4-5,6-7

36.8 ioredirect and iorestate

These two functions are used to capture the output from *stderr* or *stdout* into a file.

int ioredirect(string filename,bool err);
This function redirects either *stderr (if err is true) or stdout (is err is false)* to *filename*. It returns an *id*, which will be used to set the output back to normal.

iorestate(int id,err);
This function brings the output back to normal. The first parameter is the “id” that was returned by **ioredirect**. The file is then closed.

Example:

```
int o=ioredirect('C:\Tamgu\test\test.txt',true);
```

```
printlnerr("This string is now stored in file: test.txt");
iorestate(o,true); //back to normal
```

36.9 pause and sleep

These two functions are used either to put a thread in pause or in sleep mode. *pause* does not suspend the execution of a thread, while *sleep* does it.

pause takes as input a float, whose value is in *seconds*. Pause can take a second Boolean parameter to display a small animation.

sleep is based on the OS *sleep* instruction and its behavior depends on its local implementation. It takes as input an integer.

Example:

```
pause(0.1); the thread will pause for 10 ms
pause(2,true); the thread will pause for 2s, with a small animation
sleep(1); the thread will sleep for 1s (depending on the platform)
```

36.10 Emojis: emojis()

This procedure returns a *mapls* list of all emojis characters according to the norm *v5.0* beta.

36.11 GPSdistance(float Longitude1, float latitude1, float Longitude2, float latitude2, float radius)

Tamgu provides a method to compute the distance between two GPS points, given as longitude/latitude. The fifth parameter is optional and represents the Earth radius, whose default value is *6371* km. This value also defines, which unit will be used to compute the final value.

Example:

```
float d= GPSdistance(46.9001,7.1201, 47.01, 6.1);

        d is 78.382 km
```

36.12 Read from keyboard: kget()

Tamgu also provides a specific function *kget(),* which is used to read a string from the keyboard.

Example

```
string message=kget("Message:");
```

36.13 Persistent Variables

You can create persistent variables in the context of a GUI, in order to keep tracks of certain values when launching your programs over and over again.

Persistent types: ithrough, fthrough, uthrough, sthrough, vthrough, mthrough

You can declare a variable with one these types to keep track of different experiments. These variables are never *reinitialized* between runs.

Example

```
//This variable will keep track of the number of time this program was run
ithrough icount;
icount+=1;
println(icount);
```

37 Random

37.1 Random number: random(), a_random()

Tamgu provides a function to return a random value, which is between 0 and 99. *random()* returns a *long* value. You can also provide a maximum boundary value as an argument. *a_random* is a lock free version of random.

Example:

```
float rd=random(); // value between 0 and 99
rd=random(999); //value between 0 and 999
```

37.2 Distributions

The first parameter: nb defines the number of elements that are returned as vectors. If "nb==", then a single value (int or float) is returned. The values defined after the "=" are default values.

random_choice(int nb, vector v)

Tamgu provides a function that returns a list of values based on the values of v. The randomize function is based on a discrete distribution, where each value has the same probability to occur.

Example:

```
random_choice(10,["a","b","c"])

returns for instance:   ['c','a','b','a','b','c','b','c','a','a']
```

Uniform distributions

uniform_int(int nb, int a = 0, int b = max_value):

Produces random integer values i, uniformly distributed on the closed interval [a, b], that is, distributed according to the discrete probability function

Parameters *a, b*

Upper and lower bounds of the range ([a,b]) of possible values the distribution can generate. Note that the range includes both a and b (along with all the integer values in between).

b shall be greater than or equal to a (a<b).

It returns an int or an ivector of size nb.

uniform_real(int nb, float a=0, float b=1):

Constructs a uniform_real_distribution object, adopting the distribution parameters specified either by a and b.

Parameters *a, b*

Upper and lower bounds of the range ([a,b)) of possible values the distribution can generate. Note that the range includes a but not b.

b shall be greater than or equal to a (a<=b).

It returns a float or an fvector of size nb.

Bernoulli distributions

bernoulli_distribution(int nb, float probability = 0.5):

Construct bernoulli distribution

Constructs a binomial_distribution object, adopting the distribution parameters specified either by *t* and *p*

Parameters *t,p*

a) The upper bound of the range ([0,t]) of possible values the distribution can generate. This represents the number of independent Bernoulli-distributed experiments each generated value is said to simulate.

b) Probability of success. This represents the probability of success on each of the independent Bernoulli-distributed experiments each generated value is said to simulate.This shall be a value between 0.0 and 1.0 (both included).

It returns an ivector of Boolean values (0 or 1) of size nb.

binomial_distribution(int nb,int t = 1, float p = 0.5):

Construct binomial distribution
Constructs a binomial_distribution object, adopting the distribution parameters specified either by *t* and *p* or by object *parm*.

Parameters *t, p*

a) The upper bound of the range ([0,t]) of possible values the distribution can generate. This represents the number of independent Bernoulli-distributed experiments each generated value is said to simulate.

b) Probability of success. This represents the probability of success on each of the independent Bernoulli-distributed experiments each generated value is said to simulate. This shall be a value between 0.0 and 1.0 (both included).

It returns an int or an ivector of size nb.

negative_binomial_distribution(int nb, int k = 1, float p=0.5):

Construct negative binomial distribution
Constructs a negative_binomial_distribution object, adopting the distribution parameters specified either by *k* and *p*.

Parameters *k, p*

Parameter *k* of the negative binomial distribution.
This represents the number of unsuccessful trials that stops the count of successful Bernoulli-distributed experiments each generated value is said to simulate.

Probability of success.
This represents the probability of success on each of the independent Bernoulli-distributed experiments each generated value is said to simulate.
This shall be a value between 0.0 and 1.0 (both included).

It returns an int or an ivector of size nb.

geometric_distribution(int nb, float p = 0.5):

Construct geometric distribution
Constructs a geometric_distribution object, adopting the distribution parameter specified either by *p*.

Parameter *p*

Probability of success.
This represents the probability of success on each of the independent Bernoulli-distributed experiments each generated value is said to simulate.
This shall be a value between 0.0 and 1.0 (both included).

It returns an int or an ivector of size nb.

Poisson distributions

poisson_distribution(int nb, float mean = 1):

Construct Poisson distribution
Constructs a poisson_distribution object, adopting the distribution parameter specified either by *mean*.

Parameter *mean*

Expected number of events in the interval (μ).
This represents the rate at which the events being counted are observed, on average.
Its value shall be positive (μ>0).

It returns an int or an ivector of size nb.

exponential_distribution(int nb,float lambda = 1):

Construct exponential distribution
Constructs an exponential_distribution object, adopting the distribution parameters specified either by *lambda*.

Parameter *lambda*

Average rate of occurrence (λ).
This represents the number of times the random events are observed by interval, on average.
Its value shall be positive (λ>0).

It returns a float or a fvector of size nb.

gamma_distribution(int nb, float alpha = 1, float beta = 1):

Construct gamma distribution
Constructs a gamma_distribution object, adopting the distribution parameters specified either by *alpha* and *beta*.

Parameters *alpha, beta*

a) Parameter alpha (α), that defines the *shape* of the distribution. This shall be a positive value (α>0).

b) Parameter beta (β), that defines the *scale* of the distribution. This shall be a positive value (β>0).

It returns a float or a fvector of size nb.

weibull_distribution(int nb, float a = 1, float b = 1):

Construct Weibull distribution
Constructs a weibull_distribution object, adopting the distribution parameters specified either by *a* and *b*.

Parameters *a, b*

a) Distribution parameter *a*, which defines the *shape* of the distribution. This shall be a positive value (*a*>0).

b) Distribution parameter *b*, which defines the *scale* of the distribution. This shall be a positive value (*b*>0).

It returns a float or a fvector of size nb.

extreme_value_distribution(int nb, float a = 0, float b = 1):

Construct extreme value distribution
Constructs an extreme_value_distribution object, adopting the distribution parameters specified either by *a* and *b*.

Parameters *a, b*

a) Distribution parameter *a*, which defines the *location* (shift) of the distribution.

b) Distribution parameter *b*, which defines the *scale* of the distribution. This shall be a positive value (*b*>0).

It returns a float or a fvector of size nb.

Normal distributions

normal_distribution(int nb, float mean = 0, float stddev = 1):

Construct normal distribution
Constructs a normal_distribution object, adopting the distribution parameters specified either by *mean* and *stddev* or by object *parm*.

Parameters: *mean, stddev*

a) Mean of the distribution (its expected value, μ). Which coincides with the location of its peak.

b) Standard deviation (σ): The square root of variance, representing the dispersion of values from the distribution mean. This shall be a positive value ($\sigma>0$).

It returns a float or a fvector of size nb.

lognormal_distribution(int nb, float m = 0, float s = 1):

Construct lognormal distribution
Constructs a lognormal_distribution object, adopting the distribution parameters specified either by *m* and *s*.

Parameters *m, s*

a) Mean of the underlying normal distribution formed by the logarithm transformations of the possible values in this distribution.

b) Standard deviation of the underlying normal distribution formed by the logarithm transformations of the possible values in this distribution. This shall be a positive value ($s>0$).

It returns a float or a fvector of size nb.

chi_squared_distribution(int nb, float n = 1):

Construct Chi-squared distribution
Constructs a chi_squared_distribution object, adopting the distribution parameters specified either by *n* or by object *parm*.

Parameter *n*

Number of degrees of freedom, which specifies the number of

independent variables simulated by the distribution.
This shall be a positive value (*n*>0).

It returns a float or a fvector of size nb.

cauchy_distribution(int nb, float a=0, float b=1):

Construct Cauchy distribution
Constructs a cauchy_distribution object, adopting the distribution parameters specified either by *a* and *b*.

Parameters *a, b*

a) Distribution parameter *a*, which specifies the *location* of the peak (its mode).

b) Distribution parameter *b*, which defines the *scale* of the distribution. This shall be a positive value (*b*>0).

It returns a float or a fvector of size nb.

fisher_distribution(int nb, float m=1.0, float n=1.0):

Construct Fisher F-distribution

Constructs a fisher_f_distribution object, adopting the distribution parameters specified either by *m* and *n*.

Parameters *m, n*

a) Distribution parameter *m*, which specifies the *numerator's degrees of freedom*. This shall be a positive value (*m*>0).

b) Distribution parameter *n*, which specifies the *denominator's degrees of freedom*. This shall be a positive value (*n*>0).

It returns a float or a fvector of size nb.

student_distribution(int nb, float n=1.0):

Construct Student T-distribution
Constructs a student_t_distribution object, adopting the distribution parameters specified either by *n*.

Parameter *n*

Degrees of freedom.
Its value shall be positive (*n*>0).

It returns a float or a fvector of size nb.

Sampling distributions

discrete_distribution(int nb, ivector il):

The sequence in the list *il* is used as the *weights* for each integer value from 0 to (n-1), where n is the size of the initializer list.

It returns an int or an ivector of size nb

Example:
ivector iv = discrete_distribution(10, [40,10,10,40]);

piecewise_constant_distribution(int nb, fvector firstb, fvector firstw):

The values in the range defined within firstB are used as the *bounds* for the subintervals (b_i) and the sequence beginning at firstW is used as the *weights* (w_i) for each subinterval.

It returns a float or a fvector of size nb.

Example:

```
fvector intervals = [0.0, 2.0, 4.0, 6.0, 8.0, 10.0];
fvector weights = [2.0, 1.0, 2.0, 1.0, 2.0];
fvector res;
res=piecewise_constant_distribution(100,intervals,weights);
```

piecewise_linear_distribution(int nb, fvector firstb, fvector firstw):

The values in the range within firstB are used as the *bounds* for the subintervals (b_i) and the sequence beginning at firstW is used as the *weights* (w_i) for each subinterval bound.

It returns a float or a fvector of size nb.

Example:

```
fvector intervals = [0.0, 4.5, 9.0];
fvector weights = [10.0, 0.0, 10.0];

fvector res;

res=piecewise_linear_distribution(100,intervals,weights);
```

38 try, catch, raise

Try, catch and *raise* are used to handle errors.

catch can be associated with a string or an integer parameter. This variable is automatically set to *null* when the *try* bloc is evaluated. A catch without variable is also possible.

```
string s;
try {…
}
catch(s);
```

When an error is detected, then the error string or its number is passed to that specific variable.

38.1 Method

1. **raise(string s):** *raise an error with the message s. An error message should always starts with an error number on three characters: 000… this error number should be larger than 200, all of which are kept for internal KF errors. However no verification will be made by the language.*

Example

```
raise("201 My error");
```

39 Operator *in*

This operator has quite rich behavior, which is the reason we have a specific section dedicated to it. In the previous description, we have already described some possible utilization of this operator with files, vectors, maps or strings. We will now see how it can be extended to encompass also frames.

39.1 “in” within Frame

A frame can expose an *in* function, which will then be used when a *in* is applied to a frame. If a *in* is tested against a frame object without any *in* function, then a *false* value is always returned.

Example

```
//This is a first example of the use of in with a map.
map dico;
vector lst;
dico={'a':1,'b':6,'c':4,'d':6};

//Boolean test, it returns true or false
if (6 in dico)
   print("As expected","\n");

//The receiver is a list, then we return the list of indexes
lst=6 in dico;

string s;
for (s in lst)
   print("LST:",s,"\n");
```

RUN

```
As expected

LST: b

LST: d
```

As we can see on this example, the system returns some information in relation with the type of receiver.

Example with a frame

```
frame testframe {
    int i;

    //the type of the parameter can be anything
    function in(int j) {
        if (i==j)
            return(true);
        return(false);
    }
}
```

40 Functional Language: *à la* Lisp

Tamgu offers a *Lisp interpreter*, which is very close to the original language definition with two important exceptions:

A Lisp expression must be preceded with: \\.

For the rest, the language exposes the basic functions and operators of Lisp, but also allows for the use of Tamgu personal instructions.

Method:

The Lisp interpreter also exposes a new type: *lisp*, which is a variation on *vector*. This operator only exposes two methods:

- **eval(string):** *returns the evaluation of a string as a Lisp instruction.*
- **load(filename):** *load a filename and evaluate its content.*

Note*: For these two instructions, the double quote and the '\\' are useless.*

IMPORTANT: *It is possible to avoid using the "overloaded" Lisp with double quotes and "\\" in a file, if the two first characters in this file are: ().*

```
()   ← this characters trigger the pure Lisp mode

(defun somme (x y)
   (+ x y)
)

(println (somme 10 20))
(println (somme "a" "b"))
```

Basic Lisp Operators

The Lisp version in Tamgu exposes the following functions:

1. **apply:** */(apply f (a1 a2 a3…))*
2. **and**: *This operator is used in Boolean expressions*
3. **append**: *This operator is used to append lists together*
4. **atomp**: *Check if an element is an atom*
5. **body:** *return the body of a function*

6. **block**: *Allows for a block of instructions. The last instruction in the block returns its value*
7. **or**: *Boolean expression: (or cond1 cond2 cond3)*
8. **xor**: *Boolean expression, can have only two arguments.*
9. **break**: *To break from a loop*
10. **car**: *Return the first element of a list*
11. **cdr**: *Return the rest of the list (car and cdr can be combined into cadr, caddr, cddar etc.)*
12. **cond**: *Condition: This is a list, where each element is a list, whose first element is a condition:*
13. **cons**: *Construct a list from two elements: \(cons e1 e2)*
14. **consp**: *Check is the element is a list: \(consp e)*
15. **defun**: *Define a function (defun name (x y) e1 e2 e3)*
16. **_dropwhile:** *skip the elements until one does not match the condition. (_dropwhile CMP LIST): The CMP is either a quoted comparison or a lambda. (_dropwhile '(< 10) '(1 2 3 4 10 11 12)) returns: (10 11 12).*
17. **eq**: *Compare two atoms: \(eq e1 e2)*
18. **eval:** *Evaluate a string or a lisp expression.*
19. **_filter:** *apply a filter to a list : (_filter CMP LIST). The CMP is either a quoted comparison or a lambda.*
20. **_foldl,_foldr:** *apply an operation or a lambda to a list: (_foldl OP INIT LIST). OP is either an operator or a lambda with two arguments. The first argument is the accumulator and the second an element from the LIST. _foldl starts at the beginning of the list, _foldr from the end. (_foldl '+ 10 '(10 20 30)) returns: 70*
21. **_foldl1,_foldr1:** *apply an operation or a lambda to a list: (_foldl1 OP LIST). OP is either an operator or a lambda with two arguments. The accumulator for these functions is the first element from LIST (or the last for foldr) and the second an element from the LIST. _foldl1 starts at the beginning of the list, _foldr1 from the end. (_foldl1 '+ '(10 20 30)) returns: 60*

22. **for**: *loop in a list: \(for x lst e1 e2 e3)*
23. **if**: *Test a condition: \(if cond e_then e_else)*
24. **key:** *Return or set a value in a container: (key m k v)*
    When the "value" is omitted, the value pointed by k is returned.
    Otherwise, the value "v" is set.
25. **keys:** *Return an interval of values from within a container between kleft and kright: (keys m kleft kright v)*
    When the "value" is omitted, the value pointed by k is returned.
    Otherwise, the value "v" is set.
26. **label**: *Associate an expression with a label: \(label name e)*
27. **lambda**: *Function applied on v1,v2: \((lambda (x y) e1 e2 e3) v1 v2)*
28. **list**: *Create a list out of elements: \(list e1 e2 e3)*
29. **_map:** *apply an operation to a list : (_map OP LIST) The OP is either an operation or a lambda. There are four different possibilities. Note that you need to quote the incomplete operations, otherwise their evaluation might lead to an error.*
30. **negation**: *Boolean expression, negate the argument*
31. **nullp**: *Check if the value is* null : \(nullp e)
32. **numberp**: *Check if the value is a number:* \(numberp e)
33. **quote**: *Quote operator:* \(quote e) returns e
34. **return**: *Return a value from a for or a while: \(return e)*
35. **_scanl,_scanr:** *apply an operation or a lambda to a list: (_foldl OP INIT LIST). OP is either an operator or a lambda with two arguments. The first argument is the accumulator and the second an element from the LIST. _scanl starts at the beginning of the list, _scanr from the end. Scan works as fold, but returns a list of intermediate calculus. (_scanl '+ 10 '(10 20 30)) returns: (10 20 40 70)*
36. **_scanl1,_scanr1:** *apply an operation or a lambda to a list: (_scanl1 OP LIST). OP is either an operator or a lambda with two arguments. The accumulator for these functions is the first element from LIST (or the last for scanr1) and the second an element from the LIST. _scanl1*

starts at the beginning of the list, _scanr1 from the end. (_scanl1 '+ '(10 20 30)) returns: (10 30 60)

37. **self**: *Loop in a lambda*
38. **setq**: *Create or modify a variable: \(setq n e)*
39. **_takewhile:** *store the elements until one does not match the condition. (_takewhile CMP LIST): The CMP is either a quoted comparison or a lambda. (_takewhile '(< 10) '(1 2 3 4 10 11 12)) returns: (1 2 3 4).*
40. **while**: *Loop: \(while cond e1 e2 e3…)*
41. **zerop**: *Check if a value is 0: \(zerop e)*
42. **_zip:** *Combine lists together: (_zip '(a b c) '(e f g)) yields ((a e) (b f) (c g)).*
43. **_zipwith:** *Combine lists together with an intermediate operation or lambda: (_zipwith '+ '(1 2) '(3 4) '(5 6)) yields: (9 12).*

Some Examples

```
//an 'apply' example
\(setq f '+)
\(apply f '(1 2 3 4))

//a 'and' example
\(and (> i 10) (< i 20))

//an 'append' example
\(append '(a b c) '(e f g))  //yields (a b c e f g)

//a 'block' example
\(block
    (print 'ok)
    (+ 10 20)
)

//a 'while' example
\(while (< i 10)
    (if (i == 5)
        (break)
    )
)

//'car, cdr' examples
\(car '(a b c))      //yields a
\(cadr '((a b c d) ef)) //yields (b c d)
```

```
//'cond' example
\(cond ((< o 10) (print o)) ((> o 100) (print o)))

//'cons' example
\(cons ‘a ‘(b c)) yields (a b c)

// a function definition
\(defun add (x y) (+x y))
\(add 10 20) yields 30


//'_filter' example
//The COMPARATOR is either a quoted comparison or a lambda:
vector v = [1..10];
println(\(_filter '(< 3) v))
println(\(_filter (lambda (x) (< x 3)) v))

//Returns the list of elements that are less than 3

//'for' Loop example
vector v = [1..10];
\(for i v (print i))

//if example
\(if (== i 10) (print i) (+ i 1))

//'key' example
\(key m “test”) //returns the value whose key is “test” in m.

//'lambda' example
int j = \((lambda (x) (+ 1 x)) 2);


//list example:
\(list ’a ’b ’(c d)) //yields (a b (c d))

//'_map' example:
vector v = [1..5];

println(\(_map '* v))  //(1 4 9 16 25)
println(\(_map '(- 1) v))  //(0 1 2 3 4)
println(\(_map '(1 -) v))  //(0 -1 -2 -3 -4)
println(\(_map (lambda (x) (* x 2)) v))  //(2 4 6 8 10)

//'self' example in a lambda:
p = \( (lambda (x) (if (< x 20) (self (+ x x))  x)) 1);

//'_dropwhile' example:
v= \(_dropwhile '(< 10)  '(1 2 3 4 10 20 30));  // v = [10,20,30]

//'_takewhile' example:
v= \(_takewhile '(< 10)  '(1 2 3 4 10 20 30));  // v = [1,2,3,4]
```

```
//'_foldl' example:
int res = \(_foldl '+ 10 '(10 20 30));  // res = 70
v= \(_scanl1 (lambda (acc x) (+ acc (* 2 x))) '(10 20 30)); //v=[10,50,110]

//'_zip example:
v = \(_zip '(10 20) '(11 21) '(12 22));  // ((10 11 12) (20 21 22))

//'_zipwith example:
v= \(_zipwith '+ '(10 20) '(11 21) '(12 22)); //[33,63]
v= \(_zipwith (lambda (x y z) (- x z (+ 1 y))) '(10 20) '(11 21) '(12 22)); //[-14,-24]
```

40.1 Pure Lisp Mode

You can easily add lisp instructions to your programs, with as a constraint, the use of the '\' in front of the expression.

However, if you want to experiment in a console, these constraints might hamper your playing with the language. We have introduced: *lispmode(bool), which triggers a specific pure lisp mode, in which none of these constraints is necessary anymore. To return to the common interpreter, you need to write: (lispmode false*).

In the case of the Tamgu shell, on Linux and Mac OS, you can trigger this mode by calling: "tamgu –lisp". Within the shell, you can also use: "lispmode" to toggle between the two modes.

The last way to trigger a pure lisp mode, is to add as the two first characters in a file: (). When a file contains these two characters, the indenter will be able to consider parentheses as expression separators, similar in this spirit to "braces".

Example:

```
()
(defun test(x) (+ x x))
```

41 Functional Language: *à la* Haskell

Tamgu supplies capabilities that are similar in a quite restrictive way to the Haskell language.
The Haskell Language is a functional language, which provides some very compact and powerful ways to express specific mathematical problems, even though the language is also usually presented as a general-purpose language.

We have added to Tamgu some of the expressiveness power of the Haskell language with a specific focus on a selected range of functions. We do not pretend that Tamgu behaves as a full Haskell compiler, but it supplies some of the interesting aspects of this language.
In the rest of this chapter, we will use the expression *Tamgu Haskell* or "TASKELL" as a way to refer to the subset of the language that was integrated into Tamgu, even though we are aware that we did not go far into the very fabric of the Haskell language.

41.1 Before starting: some new operators

Before describing the language in more details, we will present some specific operators, which have been introduced to comply with some of the most interesting aspects of *Taskell*. These operators are also available in Tamgu, but their interest really rests in the way they enrich the *Taskell* world.

Range declarations: [a..b]

To comply with the Haskell language, we have added a new way to declare a range of elements: the ".." operator.

For instance [1..10] defines the vector: **[1,2,3,4,5,6,7,8,9,10]**.

step

By default the step is 1, but it is possible to set a different step. You can either directly define it with a ":" at the end of the expression:

For instance [1..10:2] defines the vector: **[1,3,5,7,9]**.

You can also define this step by providing the next element in the definition:

For instance [1,3..10] defines the vector: **[1,3,5,7,9]**.

It also works with characters:

For instance [‘a’,’c’..’g’] defines the vector: ['a','c','e','g'].

The same vector could also be defined with: [‘a’..’g’:2]...

Infinite ranges

Taskell also provides a notion of infinite range of elements. There are two cases: you can either ignore the first element of the set of the last element:

- [1..] defines an infinite vector that starts at 1, forward: [1,2,3,4…
- [..1] defines an infinite vector that starts at 1, backward: [1,0,-1,-2,-3…

You can also use different steps:

- [1..:2] defines an infinite vector that starts at 1, forward: [1,3,5…
- [..1:2] defines an infinite vector that starts at 1, backward: [1,-1,-3…

Or

- [1,3..] defines an infinite vector that starts at 1, forward: [1,3,5…
- [..-1,1] defines an infinite vector that starts at 1, backward: [1,-1,-3…

Three new operators: &&&, |||, and ::

These three operators are used to concatenate a list of elements together or to add an element to a vector.

Merge: “&&&”

This operator is used to merge different elements into a vector. If one of the elements is not a list, it is simply merged into the current list:

```
vector v= 7 &&& 8 &&& [1,2];
println(v);
```

```
v=[7,8,1,2]
```

This operator is similar to “++” in *Taskell*. Since this operator was already defined in Tamgu, we modified it into “&&&”.

Combine: “|||” (*c1* ||| *c2)*

This operator combines the values of a container with another container, or of a string with a string. When the containers are value containers, then the operator “+” is used to add or concatenate the values of c1 with c2.

```
vector v= [1,2,3];
vector vv = [4,5,6];
println(v ||| vv); //[[1,4],[2,5],[3,6]]

ivector iv= [1,2,3];
ivector ivv = [4,5,6];
println(iv ||| ivv); //[5,7,9]


string a="abc";
string b="EFG";
println(a ||| b); //aEbFcG

map m={'a':12,"b":24};
map mm={'f':12,'a':44,'c':255};

println(m ||| mm); //{'b':24,'c':255,'f':12,'a':[12,44]}

mapsi sim={'a':12,"b":24};
mapsi simm={'f':12,'a':44,'c':255};

println(sim ||| simm); //{'b':24,'f':12,'c':255,'a':56}
```

Add: "::"

This operator is similar to the other one, but with a big difference, it merges the element into the current vector.

```
1::v     →    [1,7,8,1,2]     this is the new value of v
v::12    →    [1,7,8,1,2,12]  this is the new value of v
```

41.2 Basics

Declaring a Taskell-like instruction

All *Taskell* instructions in Tamgu should be declared between "<..>", which the internal Tamgu compiler utilizes to detect a *Taskell* formula.

Example:

```
vector v=<map (+1) [1..10]>;
```

The above instruction adds 1 to each element of the vector.

Simplest structure

The simplest structure for a *Taskell* program is simply to return a value such as:

- <1>

You can return a calculus:

- <3+1>;

In that case, the system will return one single atomic value.

Example:

- < 12+3> returns 15…

Utilization of: >, <, |, << and >>

These operators can cause some issues when used inside a *Taskell* formula, since they can confuse the compiler with opening or closing *Taskell* brackets. To avoid this problem, you need to insert these expressions between parentheses:

<x | x <- [-5..5], (x > 0) > *yields* [1,2,3,4,5]
< (x << 1) | x <- [0..5]> *yields* [0,2,4,6,8,10]
<(12|3)> *yields* 15

Iteration

The *Taskell* language provides a very convenient and efficient way to represent lists. In Tamgu, these lists are implemented into "vectors", which could then be exchanged between the different structures.

The most basic *Taskell* instruction has the following form:

- <x | x <- v, Boolean>

It returns a list as result…

Which reads as:

1. We add x to our current result list.
2. We get x by iterating into v ⇔ x <- v
3. We put a Boolean constraint, which can be omitted.

The reason why it returns a list is due to the *iteration* in the expression.

Example:

<x | x <- [-5..5], x!=0> *yields* [-5,-4,-3,-2,-1,1,2,3,4,5]

Note:

You can use the "←" character instead of "<-".
Hence, you may write the above statement as:

<x | x ← [-5..5], x!=0>

Combining

You can combine different iterators together. There is two ways to do it, either as if the two iterations were embedded one into the other, or simultaneously.

1. **Embedded**

The different iterators are separated with a ","

➢ <x+y | x <-v, y <- vv, (x+y > 10)>

2. **Simultaneous**

The different iterators are combined with a ";"

➢ <x+y | x <-v ; y <- vv, (x+y > 10)>

Example:

<x+y | x <- [1..5], y <- [1..5]> //Combined
yields [2,3,4,5,6,3,4,5,6,7,4,5,6,7,8,5,6,7,8,9,6,7,8,9,10]=25 elements…

< x+y | x <- [1..5] ; y <- [1..5]> //simultaneous

yields [2,4,6,8,10]=5 elements…

Vector pattern

You can also use vector patterns to extract element from the list, if the list is composed of sub-lists.

Example

```
vector v=[[1,"P",true],[2,"C",false],[3,"E",true]];
vector vv=<[y,t] | [y,n,t] <- v, y :> 1>;
```

yields: [[2,false],[3,true]]

Iterations in maps

Tamgu also provides a specific way to iterate among maps, which in this case is quite different from what is usually available in *Haskell* implementations.

Tamgu already provides a mechanism to iterate among maps in "*for*", with keys and values provided as a recipient to the iteration process:

```
for ({x:y} in m) ...
```

The same mechanism is used here to iterate among values in a map, but also to return specific values to the recipient map.

```
< {x:y} | {y:x} <- m>;
```

Example:

```
//we declare our map
map m={"a":1,"b":2,"c":3,"d":4};

//this map is the recipient to the Taskell expression...
//We iterate among key/value in m, and we return the same values inverted...
mapis mr=< {x:y} | {y:x} <- m>;

Result is: {1:'a',4:'d',2:'b',3:'c'}
```

41.3 Declaring a local variable

There are different ways to declare local variables in a *Taskell* expression.

let operator

You can use the "let" operator, which is used to associate a variable or a vector pattern with an expression:

- <a | let a=10>
- <a+b+c | let [a,b,c] = [1,2,3]>

"let" comes with two flavors. If the return value has been declared with a "<x |…>" then different "let" expressions can be separated one from the others with a ",":

- <a+b | let a=10,let b=20>

Using Types

In above examples, the "let" declaration allows for dynamic typing, where the expression type is defined by its content.

You can however, use a specific type if needed.

- <a+b | int a=10,int b=20>

Evaluation

If an iteration is declared in your expression, then the "let" will be reevaluated at each iteration.

- <a | x <- [1..10], let a=x*2>

"in" expressions

However, it is also possible to return a value through an "in" expression. In that case, the different declarations will share one single "let". Note however, that in this case, you cannot use a specific type…

- <let a=10,b=20 in a+b>

static variable

If a *let* variable is declared together with an iteration, then this variable is re-initialized at each iteration. The keyword *static* can be utilized in this case to initialize this variable only once.

Example:

```
//We display our string and concatenate it with x
function display(self u, int x) {
    u=u+x;
    println(u);
    return true;
}

//Here tst is static
<statique() = display(tst, i) | i <- [1..3], static string tst="val">

//Here tst is re-initialized at each iteration
<dynamique() = display(tst, i) | i <- [1..3], string tst="val">
```

```
statique();
println("------");
dynamique();
```

```
val1
val12
val123
------
val1
val2
val3
```

where operator

The “where” operator is used to declare global variables. It is placed at the end of a *Taskell* expression. Its evaluation is always done once before any other analysis.

- <a | let a=w+10, where w=20>

There might be as many declarations in a “where” as necessary. Note that each declaration should be separated with “;” or a “,” from the previous.

- <a | let a=w1*w2, where w1=20,w2=30>

Note

You can also declare functions in a where, which will be local to that *Taskell* expression.

```
<description(l) = ("Liste="+<what l>) |
   where
         <what([]) : "empty">;
         <what([a]) : "one">;
         <what(xs): "large">>
```

```
description([])        yields empty
description([1])       yields one
description([1,2,3])   yields large
```

Guard

The *Haskell* language provides a mechanism which is very similar to a switch/case: the "guard". A guard is a succession of tests associated with an action, each test is introduced with a "|". The default value is introduced with the keyword "otherwise".

Example:

```
<imb(bmi) = | bmi <= 10 = "small" | bmi<=20 = "medium" | otherwise = "large">
```

imb(12) yields "medium" as a response.

Inserting Tamgu code: {…}

You can also insert some regular Tamgu code in the middle of your *Taskell* expressions. You only need to declare these instructions between {…}.

- <x | x <- [1..10], {println(x);}>

For instance, in this case, each value of x will be displayed while iterating in the value domain.

41.4 Functions

The *Taskell* language also provides a way to declare functions. These functions can be declared anywhere and can also be called as Tamgu functions.

How to declare a *Taskell* function?

A function is declared in the following way:

- <name(a1,a2…) = *Taskell* expression>

They can be called from a Tamgu program with: name(p1,p2…).

Example:

```
<one(x) = x+1>
int val=one(12);
val is: 15

<plusone(v) = x+1 | x <- v>
vector vect=plusone([1..10]);
vect is: [2,3,4,5,6,7,8,9,10,11]
```

Note that the function argument list declaration is different from the one that is defined in true Haskell.

Description of Argument Types

You can describe how a function can handle return and argument types with some simple rules.

You declare the argument types and the return type with the name of the function followed with the operator "::":

<MyFunc :: int -> int -> int>

The last element after the last "->" is the return type. The size of the arguments should match the size of the function.

If one of the arguments must be a function, then the type will be: *call*. Again, this is a difference with Haskell, where descriptions can be much more refined.

Important: If you provide declarations then all subsequent MyFunc implementations will inherit this same declaration.

When an argument in the list does not require a specific type, you can replace it with "_".

< Test :: int -> _ -> vector …>

In this example, the second argument does not request any specific type.

Example:

```
<Mult :: int -> _ -> float -> float >
< Mult(x,y,z) = 3x-y+z>

int j= 17+Mult(10,2,3);
println(j); // result is 48 (first definition of Mult)
```

Declarations in *Taskell* expressions

You can also add return types in *Taskell* expression, exactly in the same way as function, just before the expression itself. However, in that case the parameter is empty.

```
< :: ivector x | x <- [1..10]> //this expression will return an ivector
```

Without declarations

If you declare a *Taskell* function without a declaration, Tamgu declares each non-atomic element as a "self" variable.

Hence "plusone" declaration is equivalent to:

```
function plusone(self v) {…}
```

However, the arguments of these functions can be either atomic values (integer, float or string) or vector declarations.

Multiple declarations

It is actually possible to declare a function in more than one *Taskell* expressions. In that case, the argument list can contain atomic values. When the expression is evaluated, the parameters are tested against the arguments of the function. If there is no match, then the system tries the next declaration.

Example:

```
<fibonacci(0) = 0>
<fibonacci(1) = 1>
<fibonacci(n) = a | let a=fibonacci(n-1)+fibonacci(n-2)>
```

fibonacci(10) is 55

When "n" is 0 or 1, it matches against the first or second definition, which then returns the adequate value.

break

The "break" can be used to "fail" the current function declaration. For instance, you might want to go to the next declaration if the number of element is more than a certain value.

Example:

```
<myloop(v): if (v.size()>10) break else v[0]>
<myloop(v): v[10]>
```

<myloop([1..10])> yields 1, the list size is 10

```
<myloop([1..20])> yields 11, the list size is 20
```

case x of pattern -> result, pattern -> result… otherwise result

This instruction is very similar to a switch case, but with a big difference, it compares x to patterns and not just values. For instance, you can provide vector patterns in the list as a way to create local variables.

Example:

```
//In this case, we test each value against 1,2 and we return 12,24 or 34
vector v=<case x of 1 -> 12, 2 -> 24 otherwise 34 | x <- [1..10]>;
v is [12,24,34,34,34,34,34,34,34,34]

//we prepare a vector in which we have: [[1,2,3,4],..,[1,2,3,4]]
v=<replicate 5 [1..4]>;
v is [[1,2,3,4],[1,2,3,4],[1,2,3,4],[1,2,3,4],[1,2,3,4]]

//we match the sub-lists against vector patterns
v=<case x of [a,b] -> (a+b), [a,b,c,4] -> (a+b-c) otherwise <sum x> | x <- v>;
v is [0,0,0,0,0]
```

Iteration on list in the arguments…

Taskell can iterate on lists in a very similar way as Prolog. You can use the same operator "|" as in Prolog or you can use the ":" operator as in *Haskell* to define how the list should be split.

Example:

```
<see([ ]) = "empty">
<see([first:rest]) = [a,first] | let a = see(rest)>

see(['a'..'e']);
```

yields [[[[['empty','e'],'d'],'c'],'b'],'a']

41.5 Data structures: data

The keyword *data* is used to define a new data type. Data structures in Tamgu are mapped over frames.

```
<data Shape =
Circle float float float |
Rectangle float float float float
deriving(Show,Eq,Ord)>
```

In this example, the Circle value constructor has three fields, which take floats. So when we write a value constructor, we can optionally add some types after it and those types define the values it will contain. Here, the first two fields are the coordinates of its center, the third one its radius. The Rectangle value constructor has four fields which accept floats. The first two are the coordinates to its upper left corner and the second two are coordinates to its lower right one.

Now these fields are actually parameters. Value constructors are in fact functions that ultimately return a value of a data type.

Let's make a function that takes a shape and returns its surface.

```
<Surface :: Shape -> float>
<Surface(Circle _ _ r) = 2π×r²>
<Surface(Rectangle x y xx yy) = abs(xx-x) × abs(yy-y)>
```

The first notable thing here is the type declaration. It says that the function takes a shape and returns a float.

The next thing we notice here is that we can pattern match against constructors. We pattern matched against constructors before (all the time actually) when we pattern matched against values like [] or False or 5, only those values didn't have any fields. We just write a constructor and then bind its fields to names. Because we're interested in the radius, we don't actually care about the first two fields, which tell us where the circle is.

Deriving(Ds,Ds,…Ds)

The keyword *deriving* introduces some derivations into the new data structure. These derivations can stem from other data structures or from some predefined data structure, which bring some specific behavior to this new frame.

- Show: this data structure inserts some code to display the content of the data structure.

- Eq: this data structure inserts some code to compare objects one with the others. It defines a behavior that returns true if the objects fields are the same or false otherwise.

- Ord: this data structure inserts some code to compare objects with operators such as: <, >, <= and >=, in the same way as Eq.

Example:

We can extend these description in a quite more refined way. For instance, we could decide to replace the coordinates with a Point data structure.

```
//Fist we declare a Point data
<data Point = Point float float deriving(Show)>
//Which is then used to replace coordinates.
<data Shape = Circle Point float | Rect Point Point deriving(Show)>

//We modify our Point with the b offset
<dep :: Point -> float -> Point>
<dep(Point x y, b) = <Point (x+b) (y+b)>>

//Our movement function will modify a Shape object...
//It takes a Shape object and a float and returns a Shape object
//Note the $ operator, which replaces <..> around the last expression...
<mouve :: Shape -> float -> Shape>
<mouve(Circle p z, b) = <Circle  <dep p b> z> >
<mouve(Rect p1 p2, b) = <Rect <dep p1 b>  $ dep p2 b >>
//equivalent to: <mouve(Rect p1 p2, b) = <Rect <dep p1 b>  < dep p2 b >>>

let be= <mouve <Circle <Point 17 18> 30>  40>;
println(be); // Result is: <Circle <Point 57 58> 30>

let br= <mouve <Rect <Point 17 18> $ Point 30 40 >  40>;
println(br); //Result is: <Rect <Point 57 58> <Point 70 80>>
```

Data structure with field names (monades)

You can actually declare a data structure with field names, which will be mapped over a frame (as above).

You describe fields in a data structure between braces, each declaration is a field name followed by its type:

```
<data Person = Person {name :: string, lastname:: string, age :: int}>
```

The system will automatically implement each field as a frame variable.

It also creates as many function: *_name(), _lastname(), _age()* as there are fields in the frame. The function name is an alteration of the field name with a "_" as a prefix.

You can create a new data structure either with the same structure as as stated in the previous paragraph, or you can directly instantiate each field separately.

a) self nr=<Person "Pierre" "Jean" 46>; //same as above
b) nr=<Person {lastname = "Jean", name = "Dupont", age =40}>; //via fields

As you can see, in the examples above, when you initialize a structure via fields, the order is no longer important…

Enriching a Data Structure with Functions

You can also enrich a data structure with a function. For instance, if we want the above data structure to return a field, we can do:

```
<data Person = <Name() = name> >
```

The frame Person will now expose a *Name* function that returns the field *name.* We can now launch: *nr.Name()*…

Data structures are Frames

Actually, data structures are maps over frames, which means that a data structure such as:

```
<data Shape =
        Circle float float float |
        Rectangle float float float float
                deriving(Show)>
```

is actually mapped as:

```
frame Shape {
   frame Circle {
      float d1_Circle;
      float d2_Circle;
```

```
        float d3_Circle;

        function _initial(float a1, float a2, float a3) {
            d1_Circle=a1;
            d2_Circle =a2;
            d3_Circle =a3;
        }
        //derived thanks to Show
        function string() {
            return("<Circle "+d1_Rectangle +" "+d2_Rectangle +" "+d3_Rectangle
+">");
        }
    }
    frame Rectangle {
        float d1_Rectangle;
        float d2_Rectangle;
        float d3_Rectangle;
        float d4_Rectangle;

        function _initial(float a1, float a2, float a3, float a4) {
            d1_Rectangle =a1;
            d2_Rectangle =a2;
            d3_Rectangle =a3;
            d4_Rectangle =a4;
        }
        //derived thanks to Show
        function string() {
            return("<Rectangle "+d1_Rectangle +" "+d2_Rectangle +"
"+d3_Rectangle +d4_Rectangle +">");
        }
    }
}
```

The names d1… are created on the fly by the machine, including the *_initial* function and the *string* function (thanks to the *Show* derivation).

You can enrich these frames with new functions, if you want.

```
frame Circle {
    function D1() {return(d1_Circle);}
}
```

Here, we have added a P1 one function to the frame Circle, even though it was declared as a *Taskell* data structures.

We can now create a *Taskell* structure in our code and call it:

```
let bb=<Cercle 20 30 50>;
println(bb.D1()); //It displays: 20
```

With Field Names

If you declare your structure with field names, then these field names will replace the *d1,d2,d3* as described above.

Hence:

```
<data Person =
        Person {name :: string, lastname:: string, age :: int}
        deriving(Show)>
```

will be mapped as:

```
frame Person {
    string name;
    string lastname;
    int age;

    function _initial() {}
    function _initial(string n, string p, int a) {
        name=n;
        lastname=p;
        age=a;
    }

    function string() {
        return("<Person "+name+" "+lastname+" "+age+">");
    }
    function _name() {
        return(name);
    }

    function _lastname() {
        return(name);
    }
    function _age() {
        return(name);
    }
}
```

Note that a function has been created for each field, to access its content. Note as well, the presence of *_initial()* function without any arguments, which is used to assign values to field names at creation.

41.6 Functions in a *Taskell* expression

You can call any function or method, either *Taskell* or Tamgu in a *Taskell* expression. For instance, let's implement a simple "trim" on strings…

```
<trim1(w) = x | let x=w.trim()> //the simplest one
<trim2(w) = x | let x=<trim w>> //pure Taskell call

//We define a Tamgu function
function Trim(string c) {
    return(c.trim());
}

<trim3(w) = x | let x=Trim(w)> //Through an external function
<trim4(w) = x | let x=<Trim w>> //The same, but with a Taskell flavor
```

Note that *any method or function* can be called from within a *Taskell* expression as long as it matches the element type.

➢ <adding(v) = <sum v>>;

There is no actual difference between calling a function in a Tamgu way or using a *Taskell* expression.

Example: sorting a list

```
<fastsort([ ]) = [ ]>  //if the list is empty, we return an empty "list"
<fastsort([fv:v]) =  (mn &&& fv &&& mx) | //we merge the different sublists...
   let mn = fastsort(<a | a <- v, a<=fv>), //we apply our "sort" on the list that contains the elements smaller than fv (First Value)
   let mx = fastsort(<a | a <- v, a >fv>)>//we apply our "sort" on the list that contains the elements larger than fv
```

Haskell functions

We have departed from the actual function declaration in Haskell, where arguments are separated with a space. The main reason is the fact that "(…)" offers a better visual clue to detect these functions in a multi-paradigm language such as Tamgu.

A regular haskell function should be declared as: *adding x y = x+y*.

Handling functions

Taskell is a functional language, which means, among other things, that *Taskell* is able to take functions as input, but also can return functions as result.

Functions or methods as arguments

Taskell functions accept other functions as arguments. In that case, the type is "call". In this specific implementation, we do not allow for a refined description of this call as it is possible with actual *Taskell* implementation.

```
<Invert :: call -> string -> string>
<Invert(f,x) = <f x>>
println(Invert(reverse,"test")); //return tset
```

For instance the above function accepts a call as argument and applies this function on x.

Functions as result

Taskell functions can return functions as result.

```
//This function returns another function that divides x and y
<fonction :: call>
<renvoie :: _ -> _ -> int>
```

```
< fonction() = < renvoie(x,y) = x/y>>
//We call this function
call app= fonction();
//And applies it to 201,25
println(1,app(201,25)); //it returns 8, the function declaration implies an int

//A more complex example:

function machine(int j, int k) {
    return(3k-j);
}

//Here we return a function according to the value of x
<Choice(x) :
   case x of
             1 -> <calculus(x,y) = x-y>, //a local lambda
             2 -> <calculus(x,y) = y/x>,
             4 -> reverse, //it can be a method
             5 -> machine //or another function
             otherwise <calculus(x,y) = x+2y>
>

call app=Choice(1);
println(2,app(20,25)); // return -5
app=Choice(2);
println(3,app(20,25)); //returns 1.25
app=Choice(3);
println(3,app(20,25)); //returns 70

//Below the function that is applied is returned by Choice
println(4,<<Choice 2> 123 456>); // return 3.70732 (it calls calculus)
println(5,<<Choice 4> "abcdef">); //return fedcba (it calls reverse)
println(6,<<Choice 5> 123 456>); //return 1245 (it calls machine)
```

Note how function declarations influence the final value…

41.7 Maybe/Nothing

Maybe in Haskell provides a specific case to handle errors and inconsistencies. The actual definition of *Maybe* is much more complex than our implementation, which is a way to handle exceptions or errors within Taskell formalism. When, a field has been denoted as *Maybe,* then the system will automatically issue a *Nothing* as return value.

Example:

```
<division :: float → float → Maybe float>
<division(x,y) : x/y>

println(division(5,10));
println(division(5,0)); → display Nothing, as x/0 raises an exception

if (division(5,0) == Nothing)
   println("Yep");
```

41.8 Operations

Taskell provides a set of specific operations, which can only be used in *Taskell* expressions. These operations are used to apply functions or methods to a list or to filter specific values out. Other operations are used to duplicate or to cycle in a list.

<take nb list>

This function gives you the first n elements of the list. For instance, when you loop in an infinite list, this function can be used to stop the iteration after a certain number of elements have been extracted.

Example

```
<take 10 [1,5..100]> yields [1,5,9,13,17,21,25,29,33,37]
```

<drop nb list>

This gives you everything back except the first n elements of a list.

Example

```
<drop 10 [1,5..100]> yields [41,45,49,53,57,61,65,69,73,77,81,85,89,93,97]
```

<cycle list>

This method is used to cycle in a list. It can be combined with a "take" in order to limit the number of cycles.

Example

v=<take 10 <cycle [1,2,3]>> yields [1,2,3,1,2,3,1,2,3,1]

<repeat value>

This function creates a list, in which "value" is repeated *ad infinitum*. Again, you can combine it with a "take" to limit the number of iterations.

Example

v=<take 10 <repeat 5>> yields [5,5,5,5,5,5,5,5,5,5]

<replicate nb value>

This function duplicates "value" in a list of nb elements.

Example

<replicate 3 [10]> yields [[10],[10],[10]]

Composition: "."

You have certainly noted that when we apply "take" on a list, which is created through another function, we can control the number of elements that this inner function generates: **<take 10 <repeat 5>>.** This operation is called "composition". It allows for a program to put a limit on what the sub-calls are doing.

To simplify the way these compositions are written, you can use the composition operator: "."

Hence, the formula **<take 10 <repeat 5>>** can also be written as:

<take 10 .repeat 5>

<map (op) list>

This function is used to apply an operation or a function to each element of a list. If "op" is reduced to an operator, then each element is combined with itself with this operator. You can also use lambda expressions of the form (\x -> …)

With one operator

If a single operator is supplied, then it is applied to each element together.

<map (+) [1..10]> *yields* [2,4,6,8,10,12,14,16,18,20]
(1+1/2+2/3+3/4+4/5+5/6+6/7+7/8+8/9+9/10+10)

With an operator and a value

In that case, we provide both an operator and a value. Note that the position of the value in the expression is important.

<map (-1) [1..10]> *yields* [0,1,2,3,4,5,6,7,8,9]
(1-1/2-1/3-1/…/10-1)

On the other hand:

<map (1-) [1..10]> yields [0,-1,-2,-3,-4,-5,-6,-7,-8,-9]
(1-1/1-2/1-3/1-4/…1-10)

With a lambda

A lambda in *Taskell* is defined as: (\x$_0$ x$_1$..x$_n$ -> x$_0$+x$_1$+..+x$_n$), where x$_0$ x$_1$..x$_n$ are the arguments of the lambda, followed by "->" and a specific calculus.

In the case of a map, the lambda has only one argument.

<map (\x -> (x+4)/3) [1..10]> *yields* [1,2,2,2,3,3,3,4,4,4]

With a function

You can also apply a function to each element of the list, with a map.

<map (cos) [0,0.1..0.4]> *yields* [1,0.995004,0.980067,0.955336,0.921061]

This function can also be defined as a Tamgu function or as a *Taskell* function.

```
function Min(float y) {
    return(y-1);
}

<map (Min) [1..10]>  yields [0,1,2,3,4,5,6,7,8,9]
```

<filter (condition) list>

filter is used to filter each element from a list corresponding to a specific property. This property can be expressed with a comparison operator, with a lambda or with a function that returns *true* or *false.*

Examples

1. *A very simple example, we only keep values > 3:*

```
<filter (>3) [1..10]> yields [4,5,6,7,8,9,10]
```

2. *In the next example, we use a lambda expression, which is a bit richer than a simple operator. In our example, we only keep the even values.*

```
<filter (\x -> (x%2)==0) [1..10]> yields [2,4,6,8,10]
```

3. *The following example returns the list of prime numbers among the 1000 first integers. factors returns the list of dividers for a given number:*

```
<filter (\x -> <size <factors x>> ==1) [0..1000]>
```

4. *We can also use a function to do the comparison:*

```
function odd(int x) {
   if (x%2==0)
      return(false);
   return(true);
}

<filter (odd) [1..10]> yields [1,3,5,7,9]
```

5. *Finally, we can also compose our expression with a "map"*

```
<filter (odd) . map (*3) [1..10]> yields [3,9,15,21,27]
```

<all (condition) list>

This instruction returns *true* if *condition* is verified for each element.

<all (odd) [1,3,5]> yields *true*

<any (condition) list>

This instruction returns *true* if *condition* is verified for at least one element.

<any (odd) [2,3,6]> yields *true*

<and list>

This instruction returns *true* if each element is *true*.

<and [1,1,1]> yields *true*

<or list>

This instruction returns *true* if at least one element is *true*.

<or [1,0,0]> yields *true*

<xor list>

This instruction returns *true* if at least one element is *true*, but not all are *true.*

<xor [1,0,0]> yields *true*
<xor [1,1,1]> yields *false*

<takeWhile (condition) list>

takeWhile put a condition on each element from the list. When this condition *is not met* then the iteration on the list stops. It works in a similar way as "take", but instead of counting the elements, it put a condition on them.

Examples

1. *Iterating on an infinite list*

   <takeWhile (<100) [1,11..]> yields [1,11,21,31,41,51,61,71,81,91]

   As we can see on this example, the iteration has stopped when the value returned from the list is above 100…

2. *Combining with a map and a filter*

   In this example, we extract all the squares below 500 that are odd.

   <filter (odd) . takeWhile (<500) . map (*) [1..]>

The result is: [1,9,25,49,81,121,169,225,289,361,441]

<dropWhile (condition) list>

This function drops all the elements until an element does not match the condition. It then keeps the rest of the list.

Example

<dropWhile (isdigit) “12345ABCD123” > yields ABCD123

<zip l1 l2..ln>

Combine different lists together. Each element from l1,..,ln is stored into a list.

Examples

<zip [0..2] [0..2] [0..2]>

The result is: [[0,0,0],[1,1,1],[2,2,2]]

<zipWith (f) l1 l2 l3…ln>

zipWith combines different list together thanks to *f*. If *f* is a lambda, then it should have as many arguments as the number of lists in the expression.

Examples

1. *Combining three lists together with “+”*

   <zipWith (+) [0..10] [0..10] [0..10]>

   The result is: [0,3,6,9,12,15,18,21,24,27,30]

2. *With a lambda function*

   <zipWith (\x y z -> x*y+z) [0..10] [0..10] [0..10]>

   The result is: [0,2,6,12,20,30,42,56,72,90,110]

3. *Composing with a takeWhile and infinite lists*

   <takeWhile (<100) . zipWith (\x y z -> x*y+z) [0..] [0..] [0..]>

   The result is : [0,2,6,12,20,30,42,56,72,90]

<foldl|foldr (f) first list>

These operators apply a function, a lambda or an operation on a list, with "first" as a seed. The lambda function should have two arguments. The difference between "foldl" and "foldr" is the direction of the "fold". "foldl" starts from the beginning of the list, while foldr starts from the end of the list. foldl then traverses the list in a forward manner, while foldr traverses the list backward.

If you use a lambda expression, then the element from the list should be the first one for foldr and the second one for foldl. The other element is an accumulator, whose final value will be returned as a result of the *fold* expression.

The result of these functions is a value, not a list…

Examples

1. *Summing elements from a list, with as a first value 100*

   <foldl (+) 100 [1..10]> yields 155… (100+1+2+3…+10)

2. *Accumulating values in lambda expression with foldl*

   <foldl (\ acc x -> acc+2*x) 10 [1..10]> yields 120

   Note that the element from the list: "x" is the second element of the lambda expression, while the accumulator is the first one.

3. *Accumulating values in lambda expression with foldr*

   <foldr (\ x acc -> acc+2*x) 10 [1..10]> yields 120

   Note that the element from the list: "x" is the first element of the lambda expression.

<foldl1|foldr1 (f) list>

These two functions are similar to foldl and foldr, but they take as a seed the first element of the list.

Examples

1. *Summing elements from a list, the first value is 1…*

   <foldl1 (+) [1..10]> yields 55… (1+2+3…+10)

2. *Accumulating values in lambda expression with foldl1*

   <foldl1 (\ acc x -> acc+2*x) [1..10]> yields 109

   Note that the element from the list: “x” is the second element of the lambda expression.

3. *Accumulating values in lambda expression with foldr1*

   <foldr1 (\ x acc -> acc+2*x) [1..10]> yields 110

   Note that the element from the list: “x” is the first element of the lambda expression.

scanl, scanr, scanl1, scanr1

These functions are very similar to the “fold” function, except that they store in a list the intermediary results.

Examples

1. *Summing elements from a list, the first value is 1…*

   <scanl1 (+) [1..10]> yields [3,6,10,15,21,28,36,45,55]

2. *Accumulating values in lambda expression with scanl1*

   <scanl1 (\ acc x -> acc+2*x) [1..10]> yields [5,11,19,29,41,55,71,89,109]

   Note that the element from the list: “x” is the second element of the lambda expression.

3. *Accumulating values in lambda expression with scanr1*

   <scanr1 (\ x acc -> acc+2*x) [1..10]> yields [100,98,94,88,80,70,58,44,28]

   Note that the element from the list: “x” is the first element of the lambda expression.

41.9 Cosine Example

```
string s=@"
```

55512	70.7107	1 0 1 0 1 1 0 0 0	ok - so what is she attempting to download ?
56836	70.7107	1 0 0 0 1 1 0 1 0	ok do you have any data indication in the notification bar ?
80803	70.7107	1 1 0 1 0 1 0 0 0	then it appears that this device has no sd card slot.
89103	70.7107	1 0 1 0 1 1 0 0 0	well for most. there is no national data roaming charges
7203	68.6406	0 0 1 0 1 3 1 1 0	as in if you slide down the status bar to see the notifications
50244	67.082	1 2 0 0 1 2 0 0 0	no it would dot work in the s3 as the s3 uses a mini sim card
23519	66.8153	1 0 0 1 1 2 0 0 0	hi i have a new htc one ppd and the keyboard no longer?
35862	66.8153	0 0 1 0 1 2 0 1 0	if there is an update available , the phone will populate
37519	66.8153	0 0 1 0 2 1 1 0 0	in problem state , i dont see any usb related message
42803	66.8153	1 0 0 0 1 2 0 1 0	it puts it where ? so you get the icon but no sound ?
82234	66.8153	0 0 0 0 1 2 1 1 0	there are notifications in the notification bar
93971	66.8153	0 0 1 0 1 2 0 1 0	when the customer gets a new message
2056	61.2372	1 0 0 0 1 1 0 0 0	all right . well , there is an sd card slot in the phone
2161	61.2372	0 0 0 1 1 1 0 0 0	all right then i would suggest calling in our swap
2607	61.2372	1 0 0 0 1 1 0 0 0	alright , in this case , i will have to refer the customer
3083	61.2372	0 0 0 0 1 1 0 1 0	alright looks like it 's actually just a setting in this phone
3749	61.2372	0 0 0 1 1 1 0 0 0	and , in general the samsung galaxy note 7 appears
3945	61.2372	0 0 0 0 1 1 0 1 0	and do you see a data symbol in the notification bar ?
4248	61.2372	0 0 0 0 1 1 0 1 0	and in the notification bar do you see a network connection ?
4284	61.2372	0 1 0 0 1 1 0 0 0	and is there a sim card in that pjhone to save the contacts ?

"@;

```
//The dot product implementation :
<dot(v1,v2) = sum . zipWith (*) v1 v2>;
//The norm implementation :
<norm(v1) = sqrt . sum . map (*) v1>;


//norm could also be implemented as: <norm(v1) = sqrt . sum . map (\ a -> a*a) v1>;

//The cosine implementation
<cosine(v1,v2) = if (d==0) 0 else (n/d) | let n=<dot v1 v2>, let d=<norm v1>*<norm v2>>;

//We are going to split the above text into its value components. First we are only interested in
the column from 3 to 10…
//The first step is to split along carriage return (\n)
//Then we split each line along white characters:
//such as: ['55512','70.7107','1','0','1','0','1','1','0','0','0','ok','-','so'…]
//we then filter from the third column (first column is 0) to only keep one digit string…
//we get rid of the last column.
vector v;
v=< u[:-1] | line <- < <split . trim x> | x <- <split s "\n">, x.trim()!="">, let u=filter (in ['0'..'9'])
line[2:]>;


//uni contains only 1. It has the same size as one element from v.
ivector uni=<replicate <size v[0]> 1>;

ivector iv;
//We traverse our vector, converting each element into a vector of integers…
//And we compute the cosine distance between uni and these elements.
for (iv in v)
   println(iv,cosine(uni,iv));
```

42 Type fibre

A fiber is a particularly lightweight thread of execution. Like threads, fibers share address space. However, fibers use cooperative multitasking while threads use preemptive multitasking. Threads often depend on the kernel's thread scheduler to preempt a busy thread and resume another thread; fibers yield themselves to run another fiber while executing.

In Tamgu *fibers* are implemented on top of Taskell functions. These functions in order to be executed as fibers must contain an iteration, since there is no actual *yield* function.

The type *fibre* exposes the following methods:

1. **run() :** *execute the fibers which were recorded.*
2. **block():** *fibers are stored in a linked chains, which is iterated from begin to end. When a new fiber is appended to this list, it becomes the new terminal element.* Block *is a way to define the current terminal element of that list as being the iteration boundary. New fibers can still be appended but they will not be executed until* unblock *is called. If* block *is called again with a different current terminal element, then the previous boundary is moved to this new terminal element.*
3. **unblock():** unblock *releases the limit on iteration.*

Declaring a fibre

To declare a fibre, you need first to implement a Taskell function, then store this Taskell function into a fibre variable.

```
<Myfunc(a1,a2…,an) = ….>
```

```
fibre f = MyFunc;
```

To record a new fibre, you simply call it with some parameters:

```
f(p1,p2,…,pn);
```

If MyFunc does not contain an iteration, then it will be automatically executed.

If you need to store some output, you can use the stream operator.

```
vector v;
v.push(ivector());
v[-1] <<< f(200,210);
```

Ending a fibre: *break* or end of iteration

There is two ways to end a fibre, either the iterator reaches an end, or the instruction *break* is called:

<Myfunc(x,y) : if (x==10) break else (x+1)…>

Threads

Fibers can be executed within a thread. But only one fiber can execute “run” at a time in one thread. However, threads can record new fibers in parallel without any problems.

Example

```
vector v;
int i;
//We initialize a vector of integer to store the results...
for (i in <3>)
    v.push(ivector());

//-------------------------------------------------------------------
function compute(int i, int x, int y) {
    return i+x+y;
}

<myfiber(x,y) = compute(i,x,y) | i <- [x..y]>

//We initialize a fibre variable with myfiber
fibre f = myfiber;

//-------------------------------------------------------------------
//recording is done from a thread...
joined thread recording(int i, int x, int y) {
    println("Fiber:",i);
    v[i] <<< f(x,y);
}

function running() {
    f.run();
}
//-------------------------------------------------------------------

//The fibers are launched from a thread...
int n;
for (i in <0,60,20>) {
    recording(n, i, i+19);
    n++;
}

//We wait for all fibers to record
waitonjoined();

//-------------------------------------------------------------------
//We execute them...
```

```
running();

println(v);
```

Result:

```
Fiber : 0
Fiber : 2
Fiber : 1
[[19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38],[79
,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98],[139,1
40,141,142,143,144,145,146,147,148,149,150,151,152,153,154,15
5,156,157,158]]
```

43 Synchronization

Tamgu offers a simple way to put threads in a wait state. The process is very simple to put in place. Tamgu provides different functions at this effect:

43.1 Methods

1. **cast(string)**: *this instruction releases the execution of all threads waiting on string.*
2. **cast()**: *this instruction releases all threads, whatever their string state.*
3. **lock(string s)**: *this instruction put a lock on a portion of code to prevent two threads to access the same lines at the same time.*
4. **unlock(string s)**: *this instruction deletes a lock to enable other threads to get access to the content of a function.*
5. **waitonfalse(var)**: *this function put a thread in a wait state until the value of var is set to false (or zero, or anything that returns false)*
6. **waitonjoined()**: *this function waits for threads launched within the current thread to terminate. These threads must be declared with the flag join.*
7. **wait(string)** *: this function put a thread in a wait state, using a string as trigger. The wait mode is released when a cast is done on that string…*

Example 1

```
//we use the string "test" as trigger
joined thread waiting() {
   wait("test");
   println("Released");
}
//We do some job and then we release our waiting thread
```

```
joined thread counting() {
    int nb=0;

    while (nb<10000)
        nb++;

    cast("test");
    println("End counting");
}


waiting();
counting();
waitonjoined();
println("Out");
```

Execution

If we execute the program above, Tamgu will display in the following order:

```
End counting
Released
Out
```

Example 2

```
int nb=1000;

joined thread decompte() {
    while (nb>1) {
        nb--;
    }
    printlnerr("End counting",nb);
    nb--;
}

joined thread attend() {
    waitonfalse(nb);
    printlnerr("Ok");
}


attend();
decompte();

waitonjoined();
printlnerr("End");
```

43.2 Mutex: lock and unlock

There are cases when it is necessary to prevent certain threads to have access to the same lines at the same time, for instance, to force two function calls to fully apply before another thread to take control. When a *lock* is set in a given function, then the next lines of this function are no longer accessible to other threads, until an *unlock* is called.

Example

If we take the following example:

```
//We implement our thread
thread launch(string n,int m) {
    int i;
    println(n);
    //we display all our values
    for (i=0;i<m;i++)
        print(i," ");
    println();
}

function principal() {
    //we launch our thread
    launch("Premier",2);
    launch("Second",4);
}
```

If we run it, we obtain a display which is quite random, as threads execute in an undetermined order, only known to the kernel.

```
PremierSecond
00  1 1
2 3
```

This order can be imposed with locks, which will prevent the kernel from executing the same bunch of lines at the same time.

We must add *locks* into the code, to prevent the system from meshing lines in a terrible output:

```
//We re-implement our thread with a lock
thread launch(string n, int m) {
    lock("launch"); //We lock here, no one can pass anymore
    int i;
    println(n);
    //we display all our values
    for (i=0;i<m;i++)
        print(i," ");
    println();
    unlock("launch"); //We unlock with the same string, to allow passage.
}
```

Then, when we run this piece of code again, we will have a complete different output, which is more on par with what we expect:

```
Premier
0 1
Second
0 1 2 3
```

This time the lines will display according to their order in the code.

Important:

The lock strings are global to the whole code, which means that a *lock* somewhere can be *unlock* somewhere else. It also means that a *lock* on a given string might block another part of the code that would use the same string to lock its own lines. It is therefore recommended to use very specific strings to differentiate one *lock* from another.

43.3 Protected threads

The above example could have been rewritten with exactly the same behavior by a *protected* function.

```
//We re-implement our thread as a protected function
protected thread launch(string n, int m) {
   int i;
   println(n);
   //we display all our values
   for (i=0;i<m;i++)
      print(i," ");
   println();
}
```

This function will yield exactly the same output as the one above. *Protected* threads implement a *lock* at the very beginning of the execution and release it once the function is terminated. However, the advantage of using *lock* over a *protected* function is the possibility to be much more precise on which lines should be protected.

43.4 Semaphores: waitonfalse

If the above functions are useful in a multi-threaded context, there are not enough in some cases. Tamgu provides functions, which are used to synchronize threads on variable values. These functions can only be associated with simple types such as Boolean, integer, float or string.

The role of these two functions is for a *thread to wait* for a specific variable to reach a *false* value. *False* is automatically returned when a numerical variable has the value 0, when a string is empty or when a Boolean variable is set to *false.*

waitonfalse(var);

This other function *will put a thread in a wait state* until the variable *var* reaches the value *false*.

Example

```
//First we declare a variable stopby
//Important: its initial value must be different from 0
int stopby=1;

//We implement our thread
thread launch(int m) {
    //we reset stopby with the number of loops
    stopby=m;
    int i;
    //we display all our values
    for (i=0;i<m;i++) {
        print(i," ");
        //we decrement our stopby variable
        stopby--;
    }
}

function principal() {
    //we launch our thread
    launch(10);
    //we wait for stopby to reach 0...
    waitonfalse(stopby);
    println("End ");
}

principal();
```

RUN

The execution will delay the display of “END” until every single *i* has been output on screen.

```
0 1 2 3 4 5 6 7 8 9 End
```

If we remove the *waitonfalse,* the output will be rather different:

```
End 0 1 2 3 4 5 6 7 8 9
```

As we can see on this example, Tamgu will first display the message “End” before displaying any other values.

The *waitonfalse* synchronizes *principal* and *launch* together.

Note

The example above could have been implemented with *wait* and *cast* as below:

```
//We implement our thread
thread launch(int m) {
   int i;
   //we display all our values
   for (i=0;i<m;i++)
      print(i," ");
   cast("end");
}

function principal() {
   //we launch our thread
   launch(10);
   wait("end");
   println("End");
}

principal();
```

However, one should remember that only *one cast* can be performed at a time to release threads. With a *synchronous* variable, the *waitonfalse* can be triggered by different threads, not just the one that would perform a *cast.*

43.5 waitonjoined() with flag *join*

When a thread must wait for other threads to finish before carrying one, the simplest solution is to declare each of these threads as *join*, and then uses the method: *waitonjoined()*.
Different threads can wait on a different set of joined threads at the same time.

Example

```
//A first thread with a join
join thread jdisplay(string s) {
    print(s+"\r");
}

//which is launched from this thread also "join"
join thread launch(int x) {
    int i;
    for (i=0;i<5000;i++) {
        string s="Thread:"+x+"="+i;
        jdisplay(s);
    }
    //we wait our local threads to finish
    waitonjoined();
    println("End:"+x);
}

//we launch two of them
launch(0);
launch(1);
//and we wait for them to finish...
waitonjoined();
println("Termination");
```

44 Inference engine

Tamgu embeds an inference engine, which can be freely mixed with regular Tamgu instructions.

This inference engine is very similar to Prolog but with some particularities:

a) Predicates do not need to be declared beforehand, in order for Tamgu to distinguish predicates from normal functions. However, if you need to use a predicate that will be implemented later in the code, you need to declare it beforehand.

b) You do not need to declare inference variables, however, their names are very different from traditional Prolog names: they must be preceded with a "?".

c) Each inference clause finishes with a "." and not a ";"

d) Terms can be declared beforehand (as term variables). However, if you do not want to declare them, you must precede their name with a "?" as for inference variables.

e) Probabilities might be attached to predicates, which are used to choose as a first path the one with highest probabilities.

N.B. For an adequate description of Prolog language, please consults the appropriate documentations.

44.1 Types

Tamgu exposes three specific types for inference objects:

predicate

This type is used to declare predicates, which will be used in inference clauses.

This type exposes the following methods:

1. name(): return the predicate label
2. size(): return the number of arguments

3. _trace(bool): activate or deactivate the trace for this predicate, when it is the calling predicate.

term

This type is used to declare terms, which will be used in inference clauses (see the NLP example below)

Other inference types: list and associative map

- Tamgu also provides the traditional lists *à la Prolog*, which can be used with the "|" operator to handle list decomposition (see the NLP example below for a demonstration of this operator).
    - Example

```
predicate alist;

//in this clause, C is the rest of the list...
alist([?A,?B|?C],[?A,?B],?C) :- true.

v=alist([1,2,3,4,5],?X,?Y);

println(v); → [alist([1,2,3,4,5],[1,2],[3,4,5])]
```

- Tamgu also provides an associative map, which is implemented as a Tamgu map, but in which the argument order is significant.
    - Example:

```
predicate assign,avalue;


avalue(1,1) :- true.
avalue (10,2) :- true.
avalue (100,3) :- true.
avalue ("fin",4) :- true.
```

```
assign({?X:?Y,?Z:?V}) :- avalue (?X,1), avalue (?Y,2), avalue (?Z,3), avalue
(?V,4).
vector v=assign(?X);

println(v); → [assign({'100':'fin','1':10})]
```

As you can see on this example, both *keys* and *values* can depend on *inference variables.* However, the order in which these associations *key:value* are declared is important. Thus **{?X:?Y,?Z:?V}** is different from **{?Z:?V,?X:?Y }.**

predicatevar

This type is used to handle predicates to explore their names and values. A *predicatevar* can be seen as a function, whose parameters are accessible through their position in the argument list as a vector.

This type exposes the following methods*:*

1. **map()***: return the predicate as a map: ['name':name,'0':arg0,'1':arg1…]*
2. **name()***: return the predicate name*
3. **query(predicate|name,v1,v2,v3)***: build and evaluate a predicate on the fly.*
4. **remove()***: remove the predicate from memory*
5. **remove(db)***: remove the predicate from the database db*
6. **size()***: return the number of arguments*
7. **store()***: store the predicate in memory*
8. **store(db)***: store the predicate value into the database db.*
9. **vector()***: return the predicate as a vector: [name,arg0,arg1…]*

It should be noted that the method "predicate", which exists both for a map and a vector, transforms the content of a vector or a map back into a predicate as long as their content mimics the predicate output of *vector()* and *map()*.

Example:

```
vector v=[‘female’,’mary’];
predicatevar fem;

fem=v.predicate(); //we transform our vector into a predicate.
fem.store(); //we store it in the fact base.

v=fem.query(female,?X); //We build a predicate query on the fly
v=fem.query(female,’mary’); //We build a predicate query with a string
```

44.2 Clauses

A clause is defined as follow:

predicate (arg1,arg2…,argn) :- pred(arg…),pred(arg,…), etc. ;

Fact base

A fact can be declared in a program, with the following instruction:

predicate(val,val) :- true.
If you replace “*true*” with “*false*”, then this instruction is used to remove the fact from the fact base.

N.B. It is possible to replace the above expression with:

predicate(val,val).

Disjunction

Tamgu also accepts disjunctions in clauses, with the operator “;”, which can be used in lieu of “,” between predicates.

Example:

```
predicate mere,pere;
mere("jeanne","marie").
mere("jeanne","rolande").

pere("bertrand","marie").
pere("bertrand","rolande").

predicate parent;
//We then declare our rule as a disjunction...
```

```
parent(?X,?Y) :- mere(?X,?Y);pere(?X,?Y).
parent._trace(true);

vector v=parent(?X,?Y);
println(v);
```

```
Result:
r:0=parent(?X,?Y) --> parent(?X6,?Y7)
e:0=parent(?X8,?Y9) --> mere(?X8,?Y9)
 k:1=mere('jeanne','marie').
  success:2=parent('jeanne','marie')
 k:1=mere('jeanne','rolande').
  success:2=parent('jeanne','rolande')

[parent('jeanne','marie'),parent('jeanne','rolande')]
```

Cut and fail

Tamgu also provides a *cut*, which is expressed with the traditional “!”. It also provides the keyword *fail*, which can be used to force the failure of a clause.

Functions

Tamgu also provides some regular functions from the Prolog language such as:

Function asserta(pred(…))

This function asserts (inserts) a predicate at the beginning of the knowledge base. Note that this function *can only be used within a clause declaration.*

assertz(pred(…))

This function asserts (inserts) a predicate at the end of the knowledge base. Note that this function *can only be used within a clause declaration.*

retract(pred(…))

This function removes a predicate from the knowledge base. Note that this function *can only be used within a clause declaration.*

retractall(pred)

This function removes all instances of predicate “pred” from the knowledge base. If *retractall* is used without any parameters, then it cleans the whole knowledge base. Note that this function *can only be used within a clause declaration.*

Function: predicatedump(pred) or findall(pred)

This function when used without any parameters returns all predicates stored in memory as a vector. If you provide the name of a predicate as a *string,* then it dumps as a vector all the predicates with the specified name.

Example

```
//Note that you need to declare "parent" if you want to use it in an assert
predicate parent;

adding(?X,?Y) :- asserta(parent(?X,?Y)).

adding("Pierre","Roland");

println(predicatedump(parent));
```

Callback function

A predicate can be declared with a callback function, whose signature is the following:

```
function OnSuccess(predicatevar p, string s) {
   println(s,p);
   return(true);
}

string s="Parent:";

predicate parent(s) with OnSuccess;

parent("John","Mary") :- true.
parent("John","Peter") :- true.

parent(?X,?Y);
```

This function should be associated with the predicate that will be evaluated. Each time the evaluation on *parent* is successful then this function is called. The second argument in the function corresponds to the parameter given to *parent* in the declaration.

If the function returns *true*, then inference engine tries other solutions, otherwise it stops.

44.2.1.1.1 Result:

If we run our above example, we obtain:

```
Parent: parent('John','Mary')
Parent: parent('John','Peter')
```

44.3 DCG

Tamgu also accepts DCG rules (Definite Clause Grammar), with a few modifications with the original definition. First, Prolog variables should be denoted with ?V as in the other rules. Third, atoms can only be declared as strings.

Example:

```
predicate sentence,noun_phrase,verb_phrase;

term s,np,vp,d,n,v;

sentence(s(?NP,?VP)) --> noun_phrase(?NP), verb_phrase(?VP).
noun_phrase(np(?D,?N)) --> det(?D), noun(?N).
verb_phrase(vp(?V,?NP)) --> verb(?V), noun_phrase(?NP).
det(d("the")) --> ["the",?X], {?X is "big"}.
det(d("a")) --> ["a"].
noun(n("bat")) --> ["bat"].
noun(n("cat")) --> ["cat"].
verb(v("eats")) --> ["eats"].

//we generate all possible interpretations...
vector vr=sentence(?Y,[],?X);
println(vr);
```

44.4 Launching an evaluation

Evaluations are launched exactly in the same way as a function would. You can of course provide as many inference variables as you want, but you can only launch one predicate at a time, which imposes that your expression, should be first be declared as a clause if you want it to include more than one predicate.

Important

If the recipient variable is a vector, then all possible analyses will be provided. The evaluation tree will be fully traversed.

If the recipient variable is anything else, then whenever a solution is found, the evaluation is stopped.

Mapping methods to predicates.

Most object methods are mapped into predicates, in a very simple way. For instance, if a string exports the method “trim”, then a “p_trim” with two variables is created. Each method is mapped to a predicate in this fashion. For each method, we add a prefix: “p_” to transform this method into a predicate.

The *first* argument of this predicate is the head object of the method, while the *last* parameter is the result of applying this method to that object. Hence, if *s* is a string, *s.trim()* becomes *p_trim(s,?X)*, where *?X* is the result of applying trim to *s*. If ?X is unified, then the predicate will check if the *?X* is the same as *s.trim()*.

Example:

```
compute(?X,?Y) :- p_log(?X,?Y).
```

between(?X,?B,?E), succ(?X,?Y)

- between(?X,?B,?E) checks if the value ?X is between ?B and ?E.
- succ(?X,?Y) returns the successor of ?X. *succ* can also be used as term, but in that case, it only uses one argument.

Common mistakes with Tamgu variables.

If you use common variables in predicates, such as strings, integers or any other sorts of variables, you need to remember that these variables are used in predicates as comparison values. An example might clarify a little bit what we mean.

Example 1

```
string s="test";
string sx="other";
predicate comp;
comp._trace(true);

comp(s,3) :- println(s).
comp(sx,?X);
```

Execution:

```
r:0=comp(s,3) --> comp(other,?X172) --> Fail
```

This clause has failed, because *s* and *sx* have different values.

Example 2

```
string s="test";  //now they have the same values
string sx="test";
predicate comp;
comp._trace(true);

comp(s,3) :- println(s).
comp(sx,?X);
```

Execution:

```
r:0=comp(s,3) --> comp(test,?X173)
e:0=comp(test,3) --> println(s)test

 success:1=comp('test',3)
```

Be careful when you design your clauses, to use external variables as *comparison sources and not as instantiation.* If you want to pass a string value to your predicate, then the placeholder for that string should be a predicate variable.

Example 3

```
string sx="test";
predicate comp;
comp._trace(true);
```

```
comp(?s,3) :- println(?s).
comp(sx,?X);
```

Execution:

```
r:0=comp(?s,3) --> comp(test,?X176)
e:0=comp('test',3) --> println(?s177:test)test

 success:1=comp('test',3)
```

44.5 Examples

Hanoi tower

The following program solves the Hanoi tower problem for you.

```
//we declare our predicate
predicate move;

//Note the variable names, which all start with a "?"
move(1,?X,?Y,_) :-
   println('Move the top disk from ',?X,' to ',?Y).

move(?N,?X,?Y,?Z) :-
   ?N>1,
   ?M is ?N-1,
   move(?M,?X,?Z,?Y),
   move(1,?X,?Y,_),
   move(?M,?Z,?Y,?X).

//The result will be assigned to res
predicatevar res;

res=move(3,"left","right","centre");

println(res);
```

If you run this example, you obtain:

```
Move the top disk from  left  to  right
Move the top disk from  left  to  centre
Move the top disk from  right  to  centre
Move the top disk from  left  to  right
Move the top disk from  centre  to  left
Move the top disk from  centre  to  right
Move the top disk from  left  to  right
move(3,'left','right','centre')
```

Ancestor

With this program, you can find the common female ancestor between different people parent relationship.

```
//we declare all our predicates
predicate ancestor,parent,male,female,test;

//Then our clauses
ancestor(?X,?X) :- true.
ancestor(?X,?Z) :- parent(?X,?Y),ancestor(?Y,?Z).

//Our parent relations, which are stored in the fact base
parent("george","sam") :- true.
parent("george","andy") :- true.
parent("andy","mary") :- true.

male("george") :- true.
male("sam") :- true.
male("andy") :- true.

female("mary") :- true.

test(?X,?Q) :- ancestor(?X,?Q), female(?Q).
test._trace(true);

//In this case, since the recipient variable is a vector, we explore all possibilities.
vector v=test("george",?Z);
println(v);
```

Execution with a trace:

```
r:0=test(?X,?Q) --> test(george,?Z14)
```

```
  e:0=test('george',?Q16) --> ancestor('george',?Q16),female(?Q16)
   r:1=ancestor(?X,?X) --> ancestor('george',?Q16),female(?Q16)
   e:1=ancestor('george','george') --> female('george') --> Fail
   r:1=ancestor(?X,?Z) --> ancestor('george',?Q16),female(?Q16)
   e:1=ancestor('george',?Z19)                                  -->
parent('george',?Y20),ancestor(?Y20,?Z19),female(?Z19)
    k:2=parent('george','sam') --> ancestor('sam',?Z19),female(?Z19)
     r:3=ancestor(?X,?X) --> ancestor('sam',?Z19),female(?Z19)
     e:3=ancestor('sam','sam') --> female('sam') --> Fail
     r:3=ancestor(?X,?Z) --> ancestor('sam',?Z19),female(?Z19)
     e:3=ancestor('sam',?Z23)                                   -->
parent('sam',?Y24),ancestor(?Y24,?Z23),female(?Z23)
    k:2=parent('george','andy') --> ancestor('andy',?Z19),female(?Z19)
     r:3=ancestor(?X,?X) --> ancestor('andy',?Z19),female(?Z19)
     e:3=ancestor('andy','andy') --> female('andy') --> Fail
     r:3=ancestor(?X,?Z) --> ancestor('andy',?Z19),female(?Z19)
     e:3=ancestor('andy',?Z27)                                  -->
parent('andy',?Y28),ancestor(?Y28,?Z27),female(?Z27)
      k:4=parent('andy','mary') --> ancestor('mary',?Z27),female(?Z27)
       r:5=ancestor(?X,?X) --> ancestor('mary',?Z27),female(?Z27)
       e:5=ancestor('mary','mary') --> female('mary')
        success:6=test('george','mary')
       r:5=ancestor(?X,?Z) --> ancestor('mary',?Z27),female(?Z27)
       e:5=ancestor('mary',?Z31)                                -->
parent('mary',?Y32),ancestor(?Y32,?Z31),female(?Z31)
  [test('george','mary')]
```

An NLP example

This example corresponds to the clauses that have been generated out of the previous DCG grammar given as an example.

```
//We declare our predicates
predicate sentence,noun_phrase,det,noun,verb_phrase,verb;

//We also declare our terms...
term P,SN,SV,dét,nom,verbe;
sentence._trace(false);

sentence(?S1,?S3,P(?A,?B)) :- noun_phrase(?S1,?S2,?A),
verb_phrase(?S2,?S3,?B).
noun_phrase(?S1,?S3,SN(?A,?B)) :- det(?S1,?S2,?A), noun(?S2,?S3,?B).
```

```
verb_phrase(?S1,?S3,SV(?A,?B)) :- verb(?S1,?S2,?A),
noun_phrase(?S2,?S3,?B)

//Note the use of the "|" operator…
det(["the"|?X], ?X,dét("the"))  :- true.
det(["a"|?X], ?X,dét("a"))  :- true.

noun(["cat"|?X], ?X,nom("cat"))  :- true.
noun(["dog"|?X], ?X,nom("dog"))  :- true.
noun(["bat"|?X], ?X,nom("bat"))  :- true.

verb(["eats"|?X], ?X,verbe("eats"))  :- true.

vector v;

v=sentence(?X,[],?A);
println("All the sentences that can be generated:",v);

//we analyze a sentence
v=sentence(["the", "dog", "eats", "a", "bat"],[],?A);
println("The analysis:",v);
```

Execution:

```
All the sentences that can be generated:
[sentence(['the','cat','eats','the','cat'],[],P(SN(dét(the),nom(cat)),SV(verbe(eats
),SN(dét(the),nom(cat))))),sentence(['the','cat','eats','the','dog'],[],P(SN(dét(the
),nom(cat)),SV(verbe(eats),SN(dét(the),nom(dog))))),sentence(['the','cat','eat
s','the','bat'],[],P(SN(dét(the),nom(cat)),SV(verbe(eats),SN(dét(the),nom(bat))
))),sentence(['the','cat','eats','a','cat'],[],P(SN(dét(the),nom(cat)),SV(verbe(eats
),SN(dét(a),nom(cat))))),sentence(['the','cat','eats','a','dog'],[],P(SN(dét(the),no
m(cat)),SV(verbe(eats),SN(dét(a),nom(dog))))),sentence(['the','cat','eats','a','b
at'],[],P(SN(dét(the),nom(cat)),SV(verbe(eats),SN(dét(a),nom(bat))))),sentenc
e(['the','dog','eats','the','cat'],[],P(SN(dét(the),nom(dog)),SV(verbe(eats),SN(d
ét(the),nom(cat))))),sentence(['the','dog','eats','the','dog'],[],P(SN(dét(the),nom
(dog)),SV(verbe(eats),SN(dét(the),nom(dog))))),sentence(['the','dog','eats','th
e','bat'],[],P(SN(dét(the),nom(dog)),SV(verbe(eats),SN(dét(the),nom(bat))))),s
entence(['the','dog','eats','a','cat'],[],P(SN(dét(the),nom(dog)),SV(verbe(eats),
SN(dét(a),nom(cat))))),sentence(['the','dog','eats','a','dog'],[],P(SN(dét(the),no
m(dog)),SV(verbe(eats),SN(dét(a),nom(dog))))),sentence(['the','dog','eats','a',
'bat'],[],P(SN(dét(the),nom(dog)),SV(verbe(eats),SN(dét(a),nom(bat))))),sente
nce(['the','bat','eats','the','cat'],[],P(SN(dét(the),nom(bat)),SV(verbe(eats),SN(
dét(the),nom(cat))))),sentence(['the','bat','eats','the','dog'],[],P(SN(dét(the),no
m(bat)),SV(verbe(eats),SN(dét(the),nom(dog))))),sentence(['the','bat','eats','th
e','bat'],[],P(SN(dét(the),nom(bat)),SV(verbe(eats),SN(dét(the),nom(bat))))),s
entence(['the','bat','eats','a','cat'],[],P(SN(dét(the),nom(bat)),SV(verbe(eats),S
N(dét(a),nom(cat))))),sentence(['the','bat','eats','a','dog'],[],P(SN(dét(the),nom(
bat)),SV(verbe(eats),SN(dét(a),nom(dog))))),sentence(['the','bat','eats','a','bat'
],[],P(SN(dét(the),nom(bat)),SV(verbe(eats),SN(dét(a),nom(bat))))),sentence(
['a','cat','eats','the','cat'],[],P(SN(dét(a),nom(cat)),SV(verbe(eats),SN(dét(the),
nom(cat))))),sentence(['a','cat','eats','the','dog'],[],P(SN(dét(a),nom(cat)),SV(v
erbe(eats),SN(dét(the),nom(dog))))),sentence(['a','cat','eats','the','bat'],[],P(SN
(dét(a),nom(cat)),SV(verbe(eats),SN(dét(the),nom(bat))))),sentence(['a','cat','
eats','a','cat'],[],P(SN(dét(a),nom(cat)),SV(verbe(eats),SN(dét(a),nom(cat))))),
sentence(['a','cat','eats','a','dog'],[],P(SN(dét(a),nom(cat)),SV(verbe(eats),SN(
dét(a),nom(dog))))),sentence(['a','cat','eats','a','bat'],[],P(SN(dét(a),nom(cat)),
SV(verbe(eats),SN(dét(a),nom(bat))))),sentence(['a','dog','eats','the','cat'],[],P(
SN(dét(a),nom(dog)),SV(verbe(eats),SN(dét(the),nom(cat))))),sentence(['a','d
og','eats','the','dog'],[],P(SN(dét(a),nom(dog)),SV(verbe(eats),SN(dét(the),no
m(dog))))),sentence(['a','dog','eats','the','bat'],[],P(SN(dét(a),nom(dog)),SV(ve
rbe(eats),SN(dét(the),nom(bat))))),sentence(['a','dog','eats','a','cat'],[],P(SN(dé
t(a),nom(dog)),SV(verbe(eats),SN(dét(a),nom(cat))))),sentence(['a','dog','eats
','a','dog'],[],P(SN(dét(a),nom(dog)),SV(verbe(eats),SN(dét(a),nom(dog))))),se
ntence(['a','dog','eats','a','bat'],[],P(SN(dét(a),nom(dog)),SV(verbe(eats),SN(d
ét(a),nom(bat))))),sentence(['a','bat','eats','the','cat'],[],P(SN(dét(a),nom(bat)),
SV(verbe(eats),SN(dét(the),nom(cat))))),sentence(['a','bat','eats','the','dog'],[],
P(SN(dét(a),nom(bat)),SV(verbe(eats),SN(dét(the),nom(dog))))),sentence(['a'
,'bat','eats','the','bat'],[],P(SN(dét(a),nom(bat)),SV(verbe(eats),SN(dét(the),no
m(bat))))),sentence(['a','bat','eats','a','cat'],[],P(SN(dét(a),nom(bat)),SV(verbe(
eats),SN(dét(a),nom(cat))))),sentence(['a','bat','eats','a','dog'],[],P(SN(dét(a),n
om(bat)),SV(verbe(eats),SN(dét(a),nom(dog))))),sentence(['a','bat','eats','a','b
at'],[],P(SN(dét(a),nom(bat)),SV(verbe(eats),SN(dét(a),nom(bat)))))]
```

The analysis:

```
[sentence(['the','dog','eats','a','bat'],[],P(SN(dét(the),nom(dog)),SV(verbe(eats
),SN(dét(a),nom(bat)))))]
```

Animated Hanoi Tower

The code below displays an animation in which disks are moved from one column to another. It merges both graphics and predicates.

```
//we declare our predicate
```

```
predicate move;

//The initial configuration... All disks are on the left column
map columns={'left':[70,50,30],'centre':[],'right':[]};

//we draw a disk according to its position and its column
function disk(window w,int x,int y,int sz,int position) {
   int start=x+100-sz;
   int level=y-50*position;
   w.rectanglefill(start,level,sz*2+20,30,FL_BLUE);
}


function displaying(window w,self o) {

   w.drawcolor(FL_BLACK);
   w.font(FL_HELVETICA,40);

   w.drawtext("Left",180,200);
   w.drawtext("Centre",460,200);
   w.drawtext("Right",760,200);

   w.rectanglefill(200,300,20,460,FL_BLACK);
   w.rectanglefill(100,740,220,20,FL_BLACK);

   w.rectanglefill(500,300,20,460,FL_BLACK);
   w.rectanglefill(400,740,220,20,FL_BLACK);

   w.rectanglefill(800,300,20,460,FL_BLACK);
   w.rectanglefill(700,740,220,20,FL_BLACK);

   //Now we draw our disks
   vector left=columns['left'];
   vector centre=columns['centre'];
   vector right=columns['right'];
   int i;

   for (i=0;i<left;i++)
      disk(w,100,740,left[i],i+1);
   for (i=0;i<centre;i++)
```

```
        disk(w,400,740,centre[i],i+1);
    for (i=0;i<right;i++)
        disk(w,700,740,right[i],i+1);

}
window w with displaying;

//------ Inference engine part -------------
//we move from column x to y
function moving(string x,string y) {
    columns[y].push(columns[x][-1]);
    columns[x].pop();

    w.redraw();
    //a little pause after redrawing the whole stuff
    pause(0.5);
    return(true); //Important, we return true… or the predicate fails.
}



//Note the variable names, which all start with a "?"
move(1,?X,?Y,_) :- moving(?X,?Y).

move(?N,?X,?Y,?Z) :-
    ?N>1,
    ?M is ?N-1,
    move(?M,?X,?Z,?Y),
    move(1,?X,?Y,_),
    move(?M,?Z,?Y,?X).

//The inference is launched within a thread...
thread hanoi() {
    move(3,"left","right","centre");
}
//--------------------------------------------------------------
function launch(button b,self o) {
    hanoi();
}

//We put a button to lauch the inference engine
```

```
button b with launch;
w.begin(50,50,1000,800,"HANOI");
b.create(20,20,60,30,FL_Regular,FL_NORMAL_BUTTON ,"Launch");
w.end();
w.run();
```

45 dependency and synode

Dependencies are a specific linguistic object, which has become a staple of modern Natural Language Processing. Tamgu offers a specific implantation, based on the predicate engine, of these dependencies. The goal of this implementation is to take as input the analysis of a dependency parser (such as the Stanford parser) and implement some further analysis on the basis of this output.
Dependencies are then evaluated as predicates, which connect together syntactic nodes.

Our system provides to this effect a second type, the *synode*, which implement a node from a constituent tree.

45.1 synode

A synode is a syntactic node, which is defined through a feature set (implemented here as a *mapss)* and its position in the constituent tree.

A synode exposes the following methods:

1. **_initial(map m)***: Creates a syntactic node with some features.*
2. **addchild(synode)***: Add a first child node*
3. **addnext(synode)***: Add a next node*
4. **addprevious(synode)***: Add a previous node*
5. **after(synode)***: Return true if the node is after under the same parent.*
6. **attributes()***: Return the feature attributes as a vector.*
7. **before(synode)***: Return true if the node is before under the same parent.*
8. **child()***: Return the first child node or check it against the parameter.*
9. **children()***: Return the list of children for a given node or test if the node is a child.*
10. **definitions(mapss)***: Set the valid feature definitions for all 'synodes'.*

11. **last()**: *Return the last child node or check it against the parameter.*

12. **nbchildren()**: *Return the number of direct children.*

13. **next(synode)**: *Return the next node or check it against the parameter.*

14. **parent()**: *Return the parent node or check it against the parameter.*

15. **previous(synode)**: *Return the previous node or check it against the parameter.*

16. **precede(synode)**: *Return true if the node is before (anywhere in the tree)*

17. **sibling(synode)**: *test if the node is a sibling (either a sister or a descendant).*

18. **sisters(synode)**: *Return the list of sister nodes or check if the node is a sister node.*

19. **succeed(synode)**: *Return true if the node is after (anywhere in the tree)*

20. **test(string attribute)**: *Test if an attribute is part of the feature structure.*

21. **values()**: *Return the feature values as a vector.*

Creating a constituent tree

A constituent is built from the top to the bottom. When you use the function *addchild*, it adds a first child under the current node, then each call to this function, will add a new child after the previous node child.

Example

```
//we create our NP node
synode np({"pos":"np"});

//Then three lexical nodes
synode det({"pos":"det","surface":"the"});
synode adj({"pos":"adj","surface":"big"});
synode noun({"pos":"noun","surface":"dog"});

//We add them under np, one after the other.
np.addchild(det);
np.addchild(adj);
np.addchild(noun);

//We display the nodes in an indented way
function Display(synode x, int i) {
    if (x==null)
        return;
    string sp;
    sp.fill(i," ");
    println(sp,x);
    Display(x.child(),i+4);
    Display(x.next(),i);
}
Display(np,0);
```

Result:

```
#0['pos':'np']
    #0['pos':'det','surface':'the']
    #0['pos':'adj','surface':'big']
    #0['pos':'noun','surface':'dog']
```

Note the “#0”, which indicates that the synode is not a dependency variable.

45.2 Type dependency

A dependency is a relation between two synodes. You can create dependencies either directly through the type *dependency,* which can then be stored in the knowledge base with *assertz* or with a dependency rule.

A dependency is composed of a name, a feature set and a list of arguments.

Methods

It exposes the following methods:

1. **_initial([name,features,arg1,arg2..])**: *Create a dependency with a name (string), a feature set (a mapss) and a list of arguments, each of type synode.*
2. **features()**: *Return the dependency features.*
3. **name()**: *Return the dependency name.*
4. **rule()**: *Return the rule id that created this dependency*

Example:

```
//We create two lexical nodes
synode det({"pos":"det","surface":"the"});
synode noun({"pos":"noun","surface":"dog"});

dependency d(["DET",{"direct":"+"},det,noun]);

//We add it to the knowledge base
assertz(d);
println(d);
```

Result: DET['direct':'+']({"pos":"det","surface":"the"},{"pos":"noun","surface":"dog"})

Dependency Rule

A dependency rule matches the following pattern:

If ([^|~]dep[features](#x[features],#y) and/or dep(#w,#z)…)

depres(#n,#nn), …,depres(#n,#nn) / ~ / #x[..],#x[..].

Where x,y,w,z,n,nn are integers.

- Each of the "#x" are actually *synodes* that will be matched against the actual synodes in the knowledge base dependencies.
- A dependency can be preceded by "^" or "~".
- A dependency name can also be replaced with "_n", where n is an integer. You can either compare dependencies together or

compare their name. If you use one of these variables in a dependency result, then the name of the dependency recorded in the variable will be used to create this new dependency. “_n” can match any dependencies in memory as long as their arity matches.

- See below for a description of the feature structure.

NOTE: the *If* that starts such a rule should always start with a capital *“I”*, otherwise, the system will try to parse the rule as an ordinary *“if”* Boolean expression.

The rule reads:

If we have dependencies in the knowledge base that match against the one stored in the knowledge base, then we store some new dependencies using the same variable.

The rule can actually mix function calls and predicates together with the dependencies.

The “^” means that this dependency will be modified. Only one dependency, can be modified at a time in a rule.

The “~” is the negation. Before a dependency, it means that the dependency should not be present in the knowledge base.

If you replace the output of the rule with “~”, then the rule will apply, but no dependencies will be created.

Fact

The simplest way to add a dependency to the knowledge base, is to insert it as a fact.

dep[features](#1,#2).

Features

The feature structure in a dependency rule obeys some specific rules:

First, the quotes are optional around attributes and values.
Second, the “+” is the default value of any attributes with one value.

Operators

• attribute	We check the existence of the attribute
• attribute : value	The attribute is compared against value
• attribute : ~	The attribute should not have any value
• attribute ~: value	The attribute should not have the value *value*
• attribute = value	We give the attribute the value *value*
• attribute = ~	We remove the attribute from the feature set.
• attribute -: gram TREG	We compare the attribute against a (Tamgu Regular Expression)
• attribute ~-: gram	We compare the attribute against a TREG, which should fail.

_dependencies()

This method is used to trigger a dependency analysis, applying rules against the knowledge base.

_setvalidfeatures(mapss features)

This method is used to put some constraints on the valid features that can be used both for synodes and dependencies.

A feature is an attribute/value, which is mapped over a key/value structure in the map. If an attribute can take anything as a value, such as the lemma of a word, then the value should be the empty strings. The default value is "+".

Example

```
mapss feats={'Obl':'+','lemma':'','c_person':'+','CR4':'+','Punct':'+','surface':''};
_setvalidefeatures(feats);
```

45.3 Example

```
//We display the nodes in an indented way
function Display(synode x, int i) {
   if (x==null)
```

```
        return;
    string sp;
    sp.fill(i," ");
    println(sp,x);
    Display(x.child(),i+5);
    if (i) //when i==0, then it is the root of our tree, we do not want to display its sisters
        Display(x.next(),i);
}

//-----------------------------------------------------------------------------------------------------------------------
//we prepare our constituent tree
synode np1={"bar":2};
synode np2({"bar":2});
synode vp({"bar":2});
synode s({"bar":3});

synode v({"word":"eats","pers":3,"pres":"+","verb":"+"});
synode d1({"word":"the","det":"+"});
synode n1({"word":"dog","noun":"+"});
synode d2({"word":"a","det":"+"});
synode n2({"word":"bone","noun":"+"});



s.addchild(np1);
s.addchild(vp);

vp.addchild(v,np2);

np1.addchild(d1,n1);
np2.addchild(d2,n2);

//It is actually possible to add or modify existing features, as if a synode was a map
vp["pos"]="verb";
np1["pos"]="noun";
np2["pos"]="noun";
//-----------------------------------------------------------------------------------------------------------------------
//Our initial dependencies...

subj(v,n1).
obj(v,n2).
```

```
det(n1,d1).
det(n2,d2).

//We can also create it in a different way
dependency dpe([“mydep”,{},n1,n2]);

//But then we have to add it to the knowledge base ourselves
assertz(dpe);

//----------------------------------------------------------------------------------------------------------------------
//this function is called from a rule below. The #x becomes a synode.
//The function returns true, to avoid the rule to fail.
function DTree(synode n) {
   Display(n,0);
   println("---------------------------");
   return(true);
}
//----------------------------------------------------------------------------------------------------------------------
//A simple rule that inverts the nodes
If (subj(#1,#2)) inverted(#2,#1).

//A rule that uses constraints on nodes.
If (subj(#1[pres,pers:3],#2) and obj(#1,#3)) arguments(#2,#1,#3).

//We add features to a dependency
If (^subj(#1,#2)) subj[direct=+](#1,#2).

//We use _ to browse among all dependencies with two arguments, wich a constraint that two
nodes are different
If (_(#1,#2) && obj(#1,#3) && #2 != #3) link(#2,#3).

//We use dependency variables _1, and _2 to avoid creating a dependency between the same
arguments.
If (_1(#1,#2) && obj_2(#1,#3) && _1!=_2) other(#2,#3).

//we mark a node through a dependency rule, we can use some constraints into the structure as
well
If (subj(#1,#2) and obj(#1,#3) and #2[noun:+, subject=+]) ~.

//We can also write this rule, note that you need to use quotes in this case:
```

```
If (subj(#1,#2) and obj(#1,#3) and #3["object"]="+") ~.

//In this case, we access the parent of the node #1, note that parent is a synode method,
//which is available as a p_parent predicate (as for most Tamgu objects).
//We then call DTree to display it... DTree must return true, otherwise the rule will fail.
//The #3 is automaticaly transformed into a synode object when the function is called...
If (det(#1,#2) and p_parent(#1,#3) and DTree(#3)) ~.

//we use here a TREG as constraint in our rule
If (obj(#1[word -: "e%a+"],#2)) Verb(#1).

//------------------------------------------------------------------------------------------------------------------------
//we launch our dependency parser...
_dependencies();

//we gather all the dependencies in the knowledge base.
vector res=predicatedump();

//------------------------------------------------------------------------------------------------------------------------
Display(s,0);
println("----------------------------");
printjln(res);
```

Results

```
#0['bar':'2','pos':'noun']
    #0['word':'the','det':'+']
    #0['word':'dog','noun':'+','subject':'+']
----------------------------
 #0['bar':'2','pos':'noun']
    #0['word':'a','det':'+']
    #0['object':'+','word':'bone','noun':'+']
----------------------------
 #0['bar':'3']
    #0['bar':'2','pos':'noun']
        #0['word':'the','det':'+']
        #0['word':'dog','noun':'+','subject':'+']
    #0['bar':'2','pos':'verb']
        #0['word':'eats','pers':'3','pres':'+','verb':'+']
        #0['bar':'2','pos':'noun']
```

```
                #0['word':'a','det':'+']
                #0['object':'+','word':'bone','noun':'+']
  --------------------------
  other({"word":"dog","noun":"+","subject":"+"},{"object":"+","word":"bone","nou
n":"+"})
  subj['direct':'+']({"word":"eats","pers":"3","pres":"+","verb":"+"},
       {"word":"dog","noun":"+","subject":"+"})

  inverted({"word":"dog","noun":"+","subject":"+"},
              {"word":"eats","pers":"3","pres":"+","verb":"+"})

  obj({"word":"eats","pers":"3","pres":"+","verb":"+"},{"object":"+","word":"bone"
,"noun":"+"})
  det({"word":"dog","noun":"+","subject":"+"},{"word":"the","det":"+"})
  det({"object":"+","word":"bone","noun":"+"},{"word":"a","det":"+"})
  arguments({"word":"dog","noun":"+","subject":"+"},
  {"word":"eats","pers":"3","pres":"+","verb":"+"},
  {"object":"+","word":"bone","noun":"+"})
  link({"word":"dog","noun":"+","subject":"+"},{"object":"+","word":"bone","noun
":"+"})
  Verb({"word":"eats","pers":"3","pres":"+","verb":"+"})
```

46 _sys

Tamgu provides some system functionalities such as, reading the content of a directory into a vector or executing a system command. It exposes the variable: _*sys*, which should be used to access the following methods:

46.1 Methods

1. **command(string s,string outputfile)**: *execute the system command s. outputfile is optional and is used to redirect the command output (stdout). If outputfile is supplied, command also returns the content of this file as a string.*
2. **createdirectory(string path)**: *create a directory for the given path. Return false, if the directory already exists or cannot be created.*
3. **env(string var)**: *return the value of the environment variable: var*
4. **env(string var,string value)**: *set the value of the environment variable: var*
5. **listdirectory(string path)**: *return the files in a directory as a svector*
6. **ls(string path)**: *return the files in a directory as a svector*
7. **mkdir(string path)**: *create a directory for the given path Return false, if the directory already exists or cannot be created.*
8. **fileinfo(string path)**: *return a map with the following information for a given file:*
   a. *info["size"]*: size of the file
   b. *info["date"]*: date of the file
   c. *info["change"]:* date of the last change
   d. *info["access"]*: date of the last access
   e. *info["directory"]: true* if the path is a directory
   f. *info["pathname"]:* the real pathname
9. **pipe(string cmd):** *execute the command cmd and returns a svector as a result containing the output of that command.*
10. **realpath(string path):** *return the actual path for a given relative path.*
11. **getchar():** *return the last characters keyed in.* getchar *does not echo the characters on screen.*
12. **colors(int style, int code1, int code2, bool disp):** *set text color, return the color string*
13. **foregroundcolor(int color):** *set foreground text color*
14. **backgroundcolor(int color):** *set background text color*
15. **rgbforegroundcolor(int red, int green, int blue):** *set rgb foreground text color*

16. **rgbbackgroundcolor(int red, int green, int blue):** *set rgb background text color*
17. **scrollmargin(int top, int bottom):** *defines scroll margin area*
18. **deletechar(int nb):** *delete nb char*
19. **up(int nb):** *move up nb line*
20. **down(int nb):** *move down nb line*
21. **right(int nb):** *move right nb characters*
22. **left(int nb):** *move left nb characters*
23. **next_line(int nb):** *move to nb next line down*
24. **previous_line(int nb):** *move nb previous line up*
25. **column(int nb):** *move to column nb*
26. **row_column(int row, int column):** *move to row/column*
27. **home():** *move cursor to home*
28. **cls():** *clear screen and return to home position*
29. **hor_vert(int hor, int vert):** *move to hor/vert*
30. **clearscreen(int nb):** *nb=0, 1, 2, 3 for partial or full screen clearing*
31. **clear():** *clear screen*
32. **eraseline(int nb):** *nb =0, 1 or 2 for line erasement*
33. **scroll_up(int nb):** *scrolling up nb characters*
34. **scroll_down(int nb):** *scrolling down nb characters*
35. **screensizes():** *return the screen size for row and column*
36. **hasresized():** *return 'true', if screen size has changed*
37. **cursor():** *return the cursor position*
38. **inmouse():** *enable mouse tracking*
39. **outmouse():** *disable mouse tracking*
40. **ismouseaction(string key):** *return true if it is a mouse action*
41. **mousexy(string key):** *return mouse position*
42. **mousescrollup(string key):** *return mouse position when scrolling up*
43. **mousescrolldown(string key):** *return mouse position when scrolling down*
44. **mousedown1(string key):** *return mouse position when primary click*
45. **mousedown2(string key):** *return mouse position when secondary click*
46. **mouseup(string key):** *return mouse position when button up*
47. **mousetrack(string key):** *return mouse position when mouse is moving while pressed*
48. **reset():** *reset mouse mode and return to initial termios values. To be used on Unix machines when getchar has been used…*
49. **isescapesequence(string key):** *return true if key is an escape sequence*
50. **showcursor(bool show):** *show/hide cursor*

51. **resizecallback(function f):** *set the callback function that is called when the terminal window is resized. “f” is a function with two parameters, which are the new row and column sizes.*

Example

```
//This function copies all the files from a given directory to another, if they are
more recent than a given date
function cp(string thepath,string topath) {
    //We read the content of the source directory
    vector v=_sys.listdirectory(thepath);

    iterator it;
    string path;
    string cmd;
    map m;
    date t;

    //we set today's date starting at 9A.M.
    t.setdate(t.year(),t.month(),t.day(),9,0,0);

    it=v;
    for (it.begin();it.nend();it.next()) {
        path=thepath+'\'+it.value();
        //if the file if of the right type
        if (".cxx" in path || ".h" in path ||  ".c" in path) {
            m=_sys.fileinfo(path);
            //if the date is more recent than our current date
            if (m["date"]>t) {
                //we copy it
                cmd="copy "+path+' '+topath;
                println(cmd);
                //We execute our command
                _sys.command(cmd);
            }
        }
    }
}

//We call this function to copy from one directory to another
cp('C:\src','W:\src');
```

46.2 Variables

Some variables are also exposed by this library:

- _sys_keyright: *right arrow*
- _sys_keyleft: *left arrow*
- _sys_keydown: *down arrow*
- _sys_keyup: *up arrow*
- _sys_keydel: *del key*
- _sys_keyhomekey: *home key*
- _sys_keyendkey : *end key*
- _sys_keyc_up: *control+up arrown*
- _sys_keyc_down: *control+down arrown*
- _sys_keyc_right: *control+right arrown*
- _sys_keyc_left: *control+left arrown*
- _sys_keybackspace: *backspace*
- _sys_keyescape: *escape key*

For Windows, the following variables have been added:

- _sys_keyc_homekey: *control+home key*
- _sys_keyc_endkey: *control+end key*

These variables can be compared against *getchar,* which returns these values when a key is pressed.

Example

```
//we check if a character was hit
string c = _sys.getchar();
println(c.bytes()); //we display its content

//if the user presses the up arrow, we display all possible colors
if (c == _sys_keyup) {
   for (int i = 0; i < 9; i++) {
      for (int j = 0; j < 99; j++) {
         for (int k = 0; k < 99; k++) {
            _sys.colors(i,j,k);
            println("test:",i,j,k);
         }
      }
   }
}
```

Mouse Tracking

You can also track the mouse within your terminal, here is an example:

```
//We initialise the mouse tracking
_sys.inmouse();
string u;
u=_sys.getchar();
ivector iv;
while (u.ord() != 17) {
  if (_sys. ismouseaction(u)) {
     iv = _sys.mousedown1(u);
     if (iv != [])
        println("Mouse button 1 down at position:",iv);
     iv = _sys.mousexy(u);
     if (iv != [])
        println("Mouse is at position:",iv);
     iv = _sys.mousescrollup(u);
     if (iv != [])
        println("Mouse is scrolling up at position:", iv);
     iv = _sys.mousescrolldown(u);
     if (iv != [])
        println("Mouse is scrolling down at position:", iv);
    }
    u=_sys.getchar();
}

//We deactivate mouse tracking
//And reset the terminal
_sys.reset();
```

47 socket

The type *socket* handles HTML socket interactions between a client and a server.

47.1 Methods

Client Side

1. **close()**: *close the socket*
2. **close(clientid)**: *close the communication with clientid.*
3. **connect(string hostname,int port)**: *connect a client to a specific host on a specific port.*
4. **createserver(int port,int nbclients)**: *create a server on the local host with a specific port.*
5. **createserver(string hostname,int port,int nbclients)**: *create a server on a host with a specific port.*
6. **get()** *: get one character from a socket*
7. **get(int clientid)** *: get one character from a socket with clientid.*
8. **getframe(string name)**: *return a frame object remote handle of name name.*
9. **getfunction(string name)**: *return a function remote handle of name name.*
10. **gethostname()**: *return the current host name. The socket does not need to be activated to get this information.*
11. **read()**: *read a Tamgu object on the socket*
12. **read(clientid)**: *read a Tamgu object on the socket with clientid*
13. **receive(int nb)**: *read nb characters from a socket*
14. **receive(int clientid,int nb)**: *read nb characters from the socket with clientid*
15. **run(int client,string stopstring)**: *put the server in run mode. Server can now accept Remote Method Invocation (RMI) mode.*
16. **send(int clientid,string s)**: *write a simple string on the socket with clientid*
17. **send(string s)**: *write a simple string on the socket*

Server Side

18. **settimeout(int i)**: *set the timeout in seconds for both writing and reading on the socket. Use this instruction to avoid blocking on a read or on a write. A value of -1 cancels the timeout.*
19. **wait()**: *the server wait for a client to connect. It returns the client identifier, which will be used to communicate with the client.*
20. **write(clientid,o1,o2...)**: *write Tamgu objects on the socket with clientid.*
21. **write(o1,o2...)**: *write Tamgu objects on the socket*

Example: server side

```
//Server side
int clientid;
socket s; //we create a socket
string name=s.gethostname();
println("Local server:",name);
//We create our server on the socket 2020, with at most 5 connections…
s.createserver(2020,5);
//we wait for a client connection
while (true) {
   //we can accept up to 5 connections…
   clientid=s.wait();
   //we read a message from the client, it should be done in a //thread to handle
more connections.
   string message=s.read(clientid);
   message+=" and returned";
   //we write a message to the client
   s.write(clientid);
   //we close the connection
   s.close(clientid);
}
//We kill the server
s.close();
```

Example: client side

```
//Client side
socket s; //we create a socket
string name=s.gethostname();
println("Local server:",name);
//We create our server on the socket 2020
```

```
s.connect(name,2020);
//we write a message to the server
string message="Hello";
s.write(message);
//we read a message from the server
message=s.read();
println(message);
//we close the connection
s.close();
```

48 use(OS,library)

use loads dynamic compatible library in a Tamgu program, to add new functionalities, such as graphical interfaces, database management etc. The “OS” flag is optional; it can take one of the following values:

“WINDOWS” , “MACOS”, “UNIX”, “UNIX64”.

This flag is used to load specific libraries according to the platform architecture.

The *library* can be a simple name, which must match a library name stored in the directory whose path is recorded in the TAMGULIBS environment variable. *Library* can also be a full path leading to this same library.

Library Name convention

- On Unix platforms, library name are usually of the form: lib*myname*.so. To load such a library, you simple need to call: **use(“myname”);**

- On windows, library names are usually of the form: *myname*.dll. To load such a library, you simply need to call: **use(“myname”).**

It is usually more generic to write: use(“myname”), so that the code will work on all platforms without problems. However, you can use their full pathname, hence limiting the use of this code to only specific platforms. The OS flag can then be used to reinsert a little bit of generalization: ***use(“WINDOWS”,“Tamgusqlite”);***

49 Library xml: type xmldoc

This type is used to handle XML documents. It can be used to create a new XML document or to parse one. It is possible to associate a function with an xmldoc variable when parsing a document to have access to each node on the fly.

This type is accessed through the library *xml*: use("xml");

49.1 Methods

1. **close()**: *Close the current XML document and clean the memory from all XML values.*
2. **create(string topnode)**: *Create a new XML document, whose main node has topnode as name. If topnode is a full XML structure then use it to create the document...*
3. **load(string filename)**: *load an XML file*
4. **node()**: *Return the top node of the document.*
5. **onclosing(function f,myobject o)**: *Function to call when a closing tag is found (see associate function below)*
6. **parse(string buffer)**: *load an XML buffer*
7. **save(string filename,string encoding)**: *Save an XML document. If encoding is omitted, then encoding is "utf-8"*
8. **serialize(object)**: *Serialize as an XML document any Tamgu object*
9. **serializestring(object)**: *Serialize as an XML document any Tamgu object and return the corresponding string. The document is also cleaned in the process...*
10. **xmlstring()**: *return an XML document as a string.*
11. **xpath(string myxpath)**: *Evaluate an XPath and return a vector of xml nodes.*

49.2 Associated function

The associate function must have the following signature:

function xmlnode(xml n, object);

It must be declared in the following way:

```
use("xml");

xmldoc mydoc(obj) with xmlnode;
```

50 Library xml: type xml

The *xml* type exposes methods to handle XML nodes.

Important

This type is implemented as a placeholder for the *xmlNodePtr* type from the *libxml2 library* (see http://xmlsoft.org/), hence the *new* method which is necessary to get a new object for the current variable.

This type is accessed through the library *xml*: use(“xml”);

50.1 Methods

1. **child()**: *return the first child node under current node*
2. **child(xml)**: *Add an XML node as a child*
3. **content()**: *Return the content of a node*
4. **content(string n)**: *Change the content of a node.*
5. **delete()**: *delete the current internal node.*
6. **line()**: *return the line number of the current node*
7. **id()**: *return the id of the current node (only with call functions)*
8. **name()**: *return the XML node name*
9. **name(string n)**: *Change the XML node name*
10. **namespace()**: *Return the namespace of the current node as a vector.*
11. **new(string n)**: *Create a new internal node.*
12. **next()**: *return the next XML node*
13. **next(xml)**: *Add an XML node after the current node*
14. **parent()**: *return the parent node above current node*
15. **previous()**: *return the previous XML node*
16. **previous(xml)**: *Add an XML node before the current node*

17. **properties()**: *Return the properties of the XML node*

18. **properties(map props)**: *Properties are stored in map as attribute/value*

19. **root()**: *return the root node of the XML tree*

20. **xmlstring()**: *return the XML sub-tree as a string.*

21. **xmltype()** : *return the type of the XML node.*

As a string

Return the XML node name

Example

```
use(“xml”);

function Test(xml n, self nn) {
    map m=n.properties();
    println(n.name(),m,n.content());
}
xmldoc doc with Test;
doc.load("resTamgu.xml");
xml nd=doc.node();
println(nd);
while (nd!=null) {
    println(nd.content(),nd.namespace());
    nd=nd.child();
}
xmldoc nouveau;
nouveau.create("TESTAGE");
xml nd=nouveau.node();
xml n("toto");
nd.child(n);
n.new("titi");
n.content("Toto is happy");
nd.child(n);
nouveau.save("mynewfile.xml");
```

51 Library sqlite: type sqlite

Tamgu also provides a simple library to handle a SQLite database. SQlite is a very popular database system which uses simple files to handle SQL commands. If you want more information on SQLite, you will find plenty of it on the web.

The name of the library is *sqlite:* use(“sqlite”)

51.1 Methods

1. **begin()***: to enter the commit mode with DEFERRED mode.*
2. **begin(string mode)***: mode = DEFERRED|EXCLUSIVE|IMMEDIATE.*
3. **close()***: close a database*
4. **commit()***: the SQL command are then processed. It should finish a series of commands initiated with a begin.*
5. **create(x1,x2,x3)***: create a table in a database, with the arguments x1,x2...*
6. **execute(string sqlcommand)***: execute a sql command, without callback.*
7. **insert(table,column,value,...)***: insert a line in a table.*
8. **open(string pathname)***: open a database*
9. **run(string sqlcommand)***: execute a sql command with callback to store results. If the input variable is a vector, then all possible values will be stored in it. It the input variable is an iterator, then it is possible to iterate on the results of the sql command. Each result is a map, where each key is a column name.*

Example

```
//we declare a new sqlite variable
use("sqlite");

sqlite mydb;

//we open a database. If it does not exist, it creates it...
mydb.open('test.db');

try {
    //we insert a new table in the current database
    mydb.create("table1","nom TEXT PRIMARY KEY","age INTEGER");
    println("table1 est cree");
}
catch() {
    //This database already exists
    println("Already created");
}

int i;
string nm;
//We insert values in the database, using a commit mode (which is much faster)
mydb.begin();
//We insert 5000 elements
for(i=0;i<5000;i+=1) {
    nm="tiia_"+i;
    try {
        //we insert in table1 two values, one for 'nom' the other for 'age'.
        //Notice the alternation between column names and values
        mydb.insert("table1","nom",nm,"age",i);
        println(i);
    }
    catch() {
        println("Deja inseree");
    }
}
//we then commit our commands.
mydb.commit();

//we iterate among our values for a given SQL command
iterator it=mydb.run("select * from table1 where age>10;");
for (it.begin();it.end()==false;it.next())
    println("Value: ",it.value());

//We could have obtained the same result with:
//vector v=mydb.execute("select * from table1 where age>10;");
//However the risk to overflow our vector is pretty dangerous.

mydb.close();
```

52 GUI ToolKit library (libgui)

FLTK (http://www.fltk.org/) is a graphical C++ library, which has been implemented for many different platforms, ranging from Windows to Mac Os. We have embedded FLTK into a Tamgu library, in order to enrich the language with some GUI capabilities. The full range of features from FLTK has only been partially implemented into the Tamgu library. However, the available methods are enough to build simple but powerful interfaces.

To use this library: use(“gui”); at the beginning of your file.

Note

a) We have linked Tamgu with FLTK 1.3

b) The associate function methodology has been extended to most graphical objects.

52.1 Common methods

Most of the objects which are described in the next section share the following methods, which are used to handle the label associated to a window, a box, an input etc…

These methods, when used without any parameters, return their current value.

Methods

1. **align(int a)**: *define the label alignement (see below)*
2. **backgroundcolor(int color)**: *set or return the background color*
3. **coords()**: *return a vector of the widget coordinates*
4. **coords(int x,int y,int w,int h)**: *set the widget coordinates. It also accepts a vector instead of the four values.*
5. **created()**: *return true if the object has been correctly created*
6. **hide()**: *hide a widget*
7. **label(string txt)**: *set the label with a new text*
8. **labelcolor(int c)**: *set or return the font color of the label*
9. **labelfont(int f)**: *set or return the font of the label*
10. **labelsize(int i)**: *set or return the font size of the label*
11. **labeltype(int i)**: *set or return the font type of the label (see below for a description of the different types)*

12. **selectioncolor(int color)**: *set or return the widget selected color*
13. **show()**: *show a widget*
14. **timeout(float f)**: *set the timeout of an object within a thread.*
15. **tooltip(string txt)**: *associate a widget with a tooltip*

Label types

FL_NORMAL_LABEL
FL_NO_LABEL
FL_SHADOW_LABEL
FL_ENGRAVED_LABEL
FL_EMBOSSED_LABEL

Alignment

FL_ALIGN_CENTER
FL_ALIGN_TOP
FL_ALIGN_BOTTOM
FL_ALIGN_LEFT
FL_ALIGN_RIGHT
FL_ALIGN_INSIDE
FL_ALIGN_TEXT_OVER_IMAGE
FL_ALIGN_IMAGE_OVER_TEXT
FL_ALIGN_CLIP
FL_ALIGN_WRAP
FL_ALIGN_IMAGE_NEXT_TO_TEXT
FL_ALIGN_TEXT_NEXT_TO_IMAGE
FL_ALIGN_IMAGE_BACKDROP
FL_ALIGN_TOP_LEFT
FL_ALIGN_TOP_RIGHT
FL_ALIGN_BOTTOM_LEFT
FL_ALIGN_BOTTOM_RIGHT
FL_ALIGN_LEFT_TOP
FL_ALIGN_RIGHT_TOP
FL_ALIGN_LEFT_BOTTOM
FL_ALIGN_RIGHT_BOTTOM
FL_ALIGN_NOWRAP
FL_ALIGN_POSITION_MASK
FL_ALIGN_IMAGE_MASK

52.2 bitmap

This type is used to define a bitmap image that can be displayed in a window or a button. It exposes only one specific method:

Methods

1. **load (ivector bm,int w,in h)**: *load a bitmap image from a ivector, whose dimensions are w,h.*

Example

```
ivector sorceress = [
0xfc, 0x7e, 0x40, 0x20, 0x90, 0x00, 0x07, 0x80, 0x23, 0x00, 0x00, 0xc6,
   0xc1, 0x41, 0x98, 0xb8, 0x01, 0x07, 0x66, 0x00, 0x15, 0x9f, 0x03, 0x47,
   0x8c, 0xc6, 0xdc, 0x7b, 0xcc, 0x00, 0xb0, 0x71, 0x0e, 0x4d, 0x06, 0x66,
   0x73, 0x8e, 0x8f, 0x01, 0x18, 0xc4, 0x39, 0x4b, 0x02, 0x23, 0x0c, 0x04,
   0x1e, 0x03, 0x0c, 0x08, 0xc7, 0xef, 0x08, 0x30, 0x06, 0x07, 0x1c, 0x02,
   0x06, 0x30, 0x18, 0xae, 0xc8, 0x98, 0x3f, 0x78, 0x20, 0x06, 0x02, 0x20,
   0x60, 0xa0, 0xc4, 0x1d, 0xc0, 0xff, 0x41, 0x04, 0xfa, 0x63, 0x80, 0xa1,
   0xa4, 0x3d, 0x00, 0x84, 0xbf, 0x04, 0x0f, 0x06, 0xfc, 0xa1, 0x34, 0x6b,
   0x01, 0x1c, 0xc9, 0x05, 0x06, 0xc7, 0x06, 0xbe, 0x11, 0x1e, 0x43, 0x30,
   0x91, 0x05, 0xc3, 0x61, 0x02, 0x30, 0x1b, 0x30, 0xcc, 0x20, 0x11, 0x00,
   0xc1, 0x3c, 0x03, 0x20, 0x0a, 0x00, 0xe8, 0x60, 0x21, 0x00, 0x61, 0x1b,
   0xc1, 0x63, 0x08, 0xf0, 0xc6, 0xc7, 0x21, 0x03, 0xf8, 0x08, 0xe1, 0xcf,
   0x0a, 0xfc, 0x4d, 0x99, 0x43, 0x07, 0x3c, 0x0c, 0xf1, 0x9f, 0x0b, 0xfc,
   0x5b, 0x81, 0x47, 0x02, 0x16, 0x04, 0x31, 0x1c, 0x0b, 0x1f, 0x17, 0x89,
   0x4d, 0x06, 0x1a, 0x04, 0x31, 0x38, 0x02, 0x07, 0x56, 0x89, 0x49, 0x04,
   0x0b, 0x04, 0xb1, 0x72, 0x82, 0xa1, 0x54, 0x9a, 0x49, 0x04, 0x1d, 0x66,
   0x50, 0xe7, 0xc2, 0xf0, 0x54, 0x9a, 0x58, 0x04, 0x0d, 0x62, 0xc1, 0x1f,
   0x44, 0xfc, 0x51, 0x90, 0x90, 0x04, 0x86, 0x63, 0xe0, 0x74, 0x04, 0xef,
   0x31, 0x1a, 0x91, 0x00, 0x02, 0xe2, 0xc1, 0xfd, 0x84, 0xf9, 0x30, 0x0a,
   0x91, 0x00, 0x82, 0xa9, 0xc0, 0xb9, 0x84, 0xf9, 0x31, 0x16, 0x81, 0x00,
   0x42, 0xa9, 0xdb, 0x7f, 0x0c, 0xff, 0x1c, 0x16, 0x11, 0x00, 0x02, 0x28,
   0x0b, 0x07, 0x08, 0x60, 0x1c, 0x02, 0x91, 0x00, 0x46, 0x29, 0x0e, 0x00,
   0x00, 0x00, 0x10, 0x16, 0x11, 0x02, 0x06, 0x29, 0x04, 0x00, 0x00, 0x00,
   0x10, 0x16, 0x91, 0x06, 0xa6, 0x2a, 0x04, 0x00, 0x00, 0x00, 0x18, 0x24,
   0x91, 0x04, 0x86, 0x2a, 0x04, 0x00, 0x00, 0x00, 0x18, 0x27, 0x93, 0x04,
   0x96, 0x4a, 0x04, 0x00, 0x00, 0x00, 0x04, 0x02, 0x91, 0x04, 0x86, 0x4a,
   0x0c, 0x00, 0x00, 0x00, 0x1e, 0x23, 0x93, 0x04, 0x56, 0x88, 0x08, 0x00,
   0x00, 0x00, 0x90, 0x21, 0x93, 0x04, 0x52, 0x0a, 0x09, 0x80, 0x01, 0x00,
   0xd0, 0x21, 0x95, 0x04, 0x57, 0x0a, 0x0f, 0x80, 0x27, 0x00, 0xd8, 0x20,
    0x9d, 0x04, 0x5d, 0x08, 0x1c, 0x80, 0x67, 0x00, 0xe4, 0x01, 0x85, 0x04,
    0x79, 0x8a, 0x3f, 0x00, 0x00, 0x00, 0xf4, 0x11, 0x85, 0x06, 0x39, 0x08,
    0x7d, 0x00, 0x00, 0x18, 0xb7, 0x10, 0x81, 0x03, 0x29, 0x12, 0xcb, 0x00,
    0x7e, 0x30, 0x28, 0x00, 0x85, 0x03, 0x29, 0x10, 0xbe, 0x81, 0xff, 0x27,
    0x0c, 0x10, 0x85, 0x03, 0x29, 0x32, 0xfa, 0xc1, 0xff, 0x27, 0x94, 0x11,
    0x85, 0x03, 0x28, 0x20, 0x6c, 0xe1, 0xff, 0x07, 0x0c, 0x01, 0x85, 0x01,
    0x28, 0x62, 0x5c, 0xe3, 0x8f, 0x03, 0x4e, 0x91, 0x80, 0x05, 0x39, 0x40,
    0xf4, 0xc2, 0xff, 0x00, 0x9f, 0x91, 0x84, 0x05, 0x31, 0xc6, 0xe8, 0x07,
    0x7f, 0x80, 0xcd, 0x00, 0xc4, 0x04, 0x31, 0x06, 0xc9, 0x0e, 0x00, 0xc0,
    0x48, 0x88, 0xe0, 0x04, 0x79, 0x04, 0xdb, 0x12, 0x00, 0x30, 0x0c, 0xc8,
    0xe4, 0x04, 0x6d, 0x06, 0xb6, 0x23, 0x00, 0x18, 0x1c, 0xc0, 0x84, 0x04,
    0x25, 0x0c, 0xff, 0xc2, 0x00, 0x4e, 0x06, 0xb0, 0x80, 0x04, 0x3f, 0x8a,
    0xb3, 0x83, 0xff, 0xc3, 0x03, 0x91, 0x84, 0x04, 0x2e, 0xd8, 0x0f, 0x3f,
    0x00, 0x00, 0x5f, 0x83, 0x84, 0x04, 0x2a, 0x70, 0xfd, 0x7f, 0x00, 0x00,
    0xc8, 0xc0, 0x84, 0x04, 0x4b, 0xe2, 0x2f, 0x01, 0x00, 0x08, 0x58, 0x60,
    0x80, 0x04, 0x5b, 0x82, 0xff, 0x01, 0x00, 0x08, 0xd0, 0xa0, 0x84, 0x04,
    0x72, 0x80, 0xe5, 0x00, 0x00, 0x08, 0xd2, 0x20, 0x44, 0x04, 0xca, 0x02,
    0xff, 0x00, 0x00, 0x08, 0xde, 0xa0, 0x44, 0x04, 0x82, 0x02, 0x6d, 0x00,
    0x00, 0x08, 0xf6, 0xb0, 0x40, 0x02, 0x82, 0x07, 0x3f, 0x00, 0x00, 0x08,
    0x44, 0x58, 0x44, 0x02, 0x93, 0x3f, 0x1f, 0x00, 0x00, 0x30, 0x88, 0x4f,
    0x44, 0x03, 0x83, 0x23, 0x3e, 0x00, 0x00, 0x00, 0x18, 0x60, 0xe0, 0x07,
    0xe3, 0x0f, 0xfe, 0x00, 0x00, 0x00, 0x70, 0x70, 0xe4, 0x07, 0xc7, 0x1b,
    0xfe, 0x01, 0x00, 0x00, 0xe0, 0x3c, 0xe4, 0x07, 0xc7, 0xe3, 0xfe, 0x1f,
    0x00, 0x00, 0xff, 0x1f, 0xfc, 0x07, 0xc7, 0x03, 0xf8, 0x33, 0x00, 0xc0,
    0xf0, 0x07, 0xff, 0x07, 0x87, 0x02, 0xfc, 0x43, 0x00, 0x60, 0xf0, 0xff,
    0xff, 0x07, 0x8f, 0x06, 0xbe, 0x87, 0x00, 0x30, 0xf8, 0xff, 0xff, 0x07,
    0x8f, 0x14, 0x9c, 0x8f, 0x00, 0x00, 0xfc, 0xff, 0xff, 0x07, 0x9f, 0x8d,
    0x8a, 0x0f, 0x00, 0x00, 0xfe, 0xff, 0xff, 0x07, 0xbf, 0x0b, 0x80, 0x1f,
    0x00, 0x00, 0xff, 0xff, 0xff, 0x07, 0x7f, 0x3a, 0x80, 0x3f, 0x00, 0x80,
    0xff, 0xff, 0xff, 0x07, 0xff, 0x20, 0xc0, 0x3f, 0x00, 0x80, 0xff, 0xff,
    0xff, 0x07, 0xff, 0x01, 0xe0, 0x7f, 0x00, 0xc0, 0xff, 0xff, 0xff, 0x07,
    0xff, 0x0f, 0xf8, 0xff, 0x40, 0xe0, 0xff, 0xff, 0xff, 0x07, 0xff, 0xff,
    0xff, 0xff, 0x40, 0xf0, 0xff, 0xff, 0xff, 0x07, 0xff, 0xff, 0xff, 0xff,
    0x41, 0xf0, 0xff, 0xff, 0xff, 0x07];

use("gui");
bitmap b;

b.load(sorceress,75,75);
```

```
function affiche(window w,self e) {
    println("ICI");
    w.bitmap(b,FL_RED,50,50,275,275);
}

window w;

w.begin(30,30,500,500,"Test");
w.bitmap(b,FL_RED,50,50,75,75);
w.end();
w.run();
```

52.3 image

This object is used to load an image from a GIF or a JPEG file, which can then be used with a window object or a button object, through the method *image.*

Methods

1. loadjpeg(string filename): *load a JPEG image*
2. loadgif(string filename): *load a GIF image*

Utilization

Once an *image* object has been declared, you can load your file and use this object in the different *image methods (button* and *window* mainly*)* when available.

Example

```
use('gui');

image im;
im.loadjpeg(_current+"witch.jpg");

window w;

w.begin(30,30,1000,1000,"Image");
w.image(im, 100,100,500,500);
w.end();

w.run();
```

52.4 window

The *window* type is the entry point of this graphical library. It exposes many methods, which can be used to display boxes, buttons, sliders etc.

Methods

1. **alert(string msg)**: *Pop up window to display an alert*
2. **arc(float x,float y,float rad,float a1,float a2)**: *Draw an arc*
3. **arc(int x,int y,int x1, int y1, float a1, float a2)**: *Draw an arc*
4. **ask(string msg,string buttonmsg2,string buttonmsg1,…)**: *Pop up window to pose a question, return 0,1,2 according to which button was pressed up to 4 buttons.*
5. **begin(int x,int y,int w, int h,string title)**: *Create a window and begin initialisation, w and h are optional*
6. **bitmap(bitmap image,int color,int x, int y)**: *Display a bitmap at position x,y.*
7. **bitmap(bitmap image,int color,int x, int y, int w, int h)**: *Display a bitmap: x,y,w,h define the including box*
8. **border(bool b)**: *If true add or remove borders. With no parameter return if the window has borders*
9. **circle(int x,int y,int r,int color)**: *Draw a circle. 'color' is optional. It defines which color will be used to fill the circle up.*
10. **close()**: *close the window*
11. **create(int x,int y,int w, int h,string title)**: *Create a window without widgets, w and h are optional*
12. **cursor(int cursortype,int color1, int color2)**: *Set the cursor shape. See below for a list of cursor shapes.*
13. **drawcolor(int c)**: *set the color for the next drawings*
14. **drawtext(string l,int x,int y)**: *Put a text at position x,y*
15. **end()**: *end creation*
16. **flush()**: *force a redraw of all windows.*

17. **font(int f,int sz)***: Set the font name and its size*

18. **fontnumber()***: return the number of available fonts.*

19. **get(string msg)***: display a window to get a value*

20. **getfont(int num)***: get font name.*

21. **getfontsizes(int num)***: return a vector of available font sizes.*

22. **hide(bool v)***: hide the window if v is true, show it otherwise*

23. **image(image image,int x, int y, int w, int h)***: Display an image*

24. **initializefonts()***: load fonts from system. Return the number of available fonts (see below for an example)*

25. **line(int x,int y,int x1, int y1,int x2, int y2)***: Draw a line between points, x2 and y2 are optional*

26. **lineshape(int type,int width)***: Select the line shape and its thikness*

27. **lock()***: FLTK lock*

28. **menu(vector,int x,int y,int w, int h)***: initialize a menu with its callback functions*

29. **modal(bool b)***: If true make the window modal, with no parameter, it returns if the window is modal.*

30. **onclose(function,object)***: define a callback function to be called when the window is closed (see below)*

31. **onkey(int action, function,object)***: Set the callback function on a keyboard action with a given object as parameter*

32. **onmouse(int action, function,object)***: Set the callback function on a mouse action with a given object as parameter*

33. **ontime(function,float t,object o)***: define a callback function to be called every t second (see below)*

34. **password(string msg)***: Display a window to type in a password*

35. **pie(int x,int y,int x1, int y1, float a1, float a2)***: Draw a pie*

36. **plot(fvector xy,int thickness,fvector landmark)***: Plot a graph from a table of successive x,y points according to window size. If*

thickness===0 then points are continuously plotted, else it defines the diameter of the point. Return a float vector which is used with plotcoords. The landmark vector is optional, it is has the following structure: [XminWindow,YminWindow,XmaxWindow,YmaxWindow,XminValue,YminValue,XmaxValue,YmaxValue,incX,incY,thickness]. Only the two first values are mandatory, however the vector can only have the following size: 4,8,10,11.

37. **plotcoords(fvector landmark,float x,float y)**: *Compute the coordinates of a point(x,y) according to the previous scale computed with plot. Returns a vector of two elements [xs,ys] corresponding to the screen coordinates in the current window.*

38. **point(int x,int y)**: *Draw a pixel*

39. **polygon(int x,int y,int x1, int y1,int x2, int y2, int x3, int y3)**: *Draw a polygon, x3 and y3 are optional*

40. **popclip()**: *Release a clip region*

41. **position()**: *return a vector of the x,y position of the window*

42. **position(int x,int y)**: *position the window at the coordinates x,y*

43. **post(function f, …):** *used in threads to postpone the execution of a function f with its parameters in the main thread. In Tamgu every single graphical method sent from a thread is actually "posted" in order to be executed in the main thread. Each instruction is inserted into a linked list, but with no guaranty in which order they will be executed by the scheduler in the main thread. The "post" instruction pushed into the linked list a function call, which will be executed on the main thread. This function will be fully executed before yielding back to the main thread.*

44. **pushclip(int x,int y,int w, int h)**: *Insert a clip region, with the following coordinates*

45. **rectangle(int x,int y,int w, int h, int c)**: *Draw a rectangle with optional color c*

46. **rectanglefill(int x,int y,int w, int h, int c)**: *Fill a rectangle with optional color c*

47. **redraw()**: *Redraw the window*

48. **redrawing(float t)**: *Redraw a window every t time lapse*

49. **resizable(object)**: *make the object resizable*

50. **rgbcolor(int color)**: *return a vector of the color decomposition of into RGB components*

51. **rgbcolor(int r,int g,int b)**: *return the int corresponding to the combination of RGB components.*

52. **rgbcolor(vector rgb)**: *return the int corresponding to the combination of RGB components, which are stored in a vector.*

53. **rotation(float x,float y,float distance, float angle, bool draw)**: *Compute the coordinate of a rotated point from point x,y, using a distance and an angle. Return a vector of floats with the new coordinates. If draw is true then the line is actually drawn.*

54. **run()**: *Launch the GUI*

55. **scrollbar(int x, int y, int wscroll,int hscroll, int vwscroll, vhscroll)**: *Creates a scrollbar zone, of actual dimension x,y,wscroll,hscroll, but within a virtual zone up to vwscroll, vhscroll. Requires a window callback function to draw within this zone.*

56. **size()**: *return a 4 values vector of the window size*

57. **size(int x,int y,int w,int h)**: *resize the window*

58. **sizerange(int minw,int minh, int maxw,int maxh)**: *define range in which the size of the window can evolve*

59. **textsize(string l)**: *Return a map with w and h as key to denote width and height of the string in pixels*

60. **unlock()**: *FLTK unlock*

Drawing complex shape

The following instructions extends the above instructions.

The complex drawing functions let you draw arbitrary shapes with 2-D linear transformations. The functionality matches that found in the Adobe® PostScript™ language. The exact pixels that are filled are less defined than for the fast drawing functions so that FLTK can take advantage of drawing hardware. On both X and MS Windows the transformed vertices are rounded to integers before drawing the line segments: this severely limits the accuracy of these functions for complex graphics.

Save and restore the current transformation. The maximum depth of the stack is 32 entries.

1. pushmatrix()
2. popmatrix()

Concatenate another transformation onto the current one. The rotation angle is in degrees (not radians) and is counter-clockwise.

1. scale(float x,float y)
2. scale(float x)
3. translate(float x,float y)
4. rotate(float d)
5. multmatrix(float a,float b,float c,float d,float x,float y)

Transform a coordinate or a distance using the current transformation matrix. After transforming a coordinate pair, it can be added to the vertex list without any further translations using transformedvertex().

1. float transformx(float x, float y)
2. float transformy(float x, float y)
3. float transformdx(float x, float y)
4. float transformdy(float x, float y)
5. transformedvertex(float xf, float yf)

Start and end drawing a list of points. Points are added to the list with vertex().

1. beginpoints()
2. endpoints()

Start and end drawing lines.

1. beginline()
2. endline()

Start and end drawing a closed sequence of lines.

1. beginloop()
2. endloop()

Start and end drawing a convex filled polygon.

1. beginpolygon()
2. endpolygon()

Start and end drawing a complex filled polygon. This polygon may be concave, may have holes in it, or may be several disconnected pieces. Call gap() to separate loops of the path. It is unnecessary but harmless to call gap() before the first vertex, after the last one, or several times in a row.

gap() should only be called between begincomplexpolygon() and endcomplexpolygon(). To outline the polygon, use beginloop() and replace each gap() with a endloop();beginloop() pair.

1. begincomplexpolygon()
2. gap()
3. endcomplexpolygon()

Note: For portability, you should only draw polygons that appear the same whether "even/odd" or "non-zero" winding rules are used to fill them. Holes should be drawn in the opposite direction of the outside loop.

Add a single vertex to the current path.

- vertex(float x,float y)

Add a series of points on a Bezier curve to the path. The curve ends (and two of the points are) at X0,Y0 and X3,Y3.

- curve(float X0, float Y0, float X1, float Y1, float X2, float Y2, float X3, float Y3)

Add a series of points to the current path on the arc of a circle; you can get elliptical paths by using scale and rotate before calling arc(). The center of the circle is given by x and y, and r is its radius. arc() takes start and end angles that are measured in degrees counter-clockwise from 3 o'clock. If end is less than start then it draws the arc in a clockwise direction.

1. arc(float x, float y, float r, float start, float end)
2. circle(float x, float y, float r)

circle(...) is equivalent to arc(...,0,360) but may be faster. It must be the only thing in the path: if you want a circle as part of a complex polygon you must use arc().

Note: circle() draws incorrectly if the transformation is both rotated and non-square scaled.

Example:

```
use('gui');

int angle = 10;

function circling(window w, self e) {
    w.pushmatrix();
    angle+=5;
    w.drawcolor(FL_RED);
    w.beginpolygon();
    w.arc(500,500,130,130,angle);
    w.endpolygon();
    w.popmatrix();

}

window w with circling;


w.begin(30,30,1000,1000,"Circling");
w.redrawing(0.1);
w.end();
```

```
w.run();
```

onclose

It is possible to intercept the closing of a window with a special *callback* function, which should return *true*, if the action of closing the window is to be processed.

The function should have the following form:

function closing(window w,myobject o);

If this function returns *false* then the action of closing the window is stopped.

Example

```
use("gui");

function closing(window w, bool close) {
   if (close==false) {
      println("We cannot close this window");
      return(false);
   }
   return(true);
}

//We first declare our window object
window w;
bool closed=false;
//We then begin our window instanciation
w.begin(300,200,1300,150,"Modification");
w.onclose(closing,closed);
```

ontime

It is possible to define a function that is called every t^th second. This function must have the following parameters:

function timeout_callback(window w, object o);

Important:

If this function returns 0, then the clock is stopped. However, if this function returns any other float value, then the clock is reset and a new call is scheduled.

Example

```
use("gui");

//the callback function
function temps(window w,self n) {
    println("Ok");
    return(0.1);
}

window w;

w.begin(40,40,400,500,"Browsing");
w.ontime(temps,0.1,null);
w.end();
w.run();
```

Colors

Tamgultk library implements a few simple ways to select colors. Colors are implemented as *int*.

The predefined colors are the following:

FL_GRAY0
FL_DARK3
FL_DARK2
FL_DARK1
FL_LIGHT1
FL_LIGHT2
FL_LIGHT3
FL_BLACK
FL_RED
FL_GREEN
FL_YELLOW
FL_BLUE
FL_MAGENTA

FL_CYAN
FL_DARK_RED
FL_DARK_GREEN
FL_DARK_YELLOW
FL_DARK_BLUE
FL_DARK_MAGENTA
FL_DARK_CYAN
FL_WHITE

How to define your own colors…

It is also possible to define your own colors with an RGB encoding. RGB stands for Red Blue Green.

Tamgu provides the following method at this effect: **rgbcolor**.

a) **vector rgb=rgbcolor(int c)**: this method returns a vector containing the decomposition of that color c into its RGB components.
b) **int c=rgbcolor(vector rgb):** this method takes as input a vector containing the RGB encoding and returns the equivalent color.
c) **int c=rgbcolor(int r,int g,int b):** same as above, but takes the three components individually.

Each component is a value in: [0..255]...

Fonts

Tamgulitk provides the following font codes:

FL_HELVETICA
FL_HELVETICA_BOLD
FL_HELVETICA_ITALIC
FL_HELVETICA_BOLD_ITALIC
FL_COURIER
FL_COURIER_BOLD
FL_COURIER_ITALIC
FL_COURIER_BOLD_ITALIC
FL_TIMES
FL_TIMES_BOLD
FL_TIMES_ITALIC
FL_TIMES_BOLD_ITALIC
FL_SYMBOL
FL_SCREEN
FL_SCREEN_BOLD

FL_ZAPF_DINGBATS
FL_FREE_FONT
FL_BOLD
FL_ITALIC
FL_BOLD_ITALIC

Example

The following example shows how all available fonts can be loaded from the current system to enrich the list above.

```
use("gui");

window w;
map styles;
editor wo;
int ifont=0;

//we modify the current style of the editor to reflect the selected font
function fontchoice(int idfont) {
    //we create a new default style, whose font id is i
    styles["#"]=[FL_BLACK,idfont,16];
    wo.addstyle(styles);
    //we modify the title of the editor label to reflect the current font name
    wo.label(w.getfont(idfont)+":"+idfont);
    //to be sure that the label will be correctly we re-display the whole window
    w.redraw();
}


//Whenever the "next" button is pressed we change our current font
function change(button b,int i) {
    fontchoice(ifont);
    ifont++;
}



button b(ifont) with change;
```

```
w.begin(50,50,800,500,"Font Display");
w.sizerange(10,10,0,0);


int i;
//First we load our font from the system, to see which fonts are available
int nb=w.initializefonts();

wo.create(70,30,730,460,"Fonts");
//we use a default and available anywhere font
styles["#"]=[FL_BLACK,FL_HELVETICA,16];
wo.addstyle(styles);

//we loop among all available fonts to display both their name
//and their available sizes. [0] means that every size is available
string s,fonts;
vector v;
for (i=0;i<nb;i++) {
    if (fonts!="")
        fonts+="\r";
    s=w.getfont(i);
    v=w.getfontsizes(i);
    fonts+=i+":"+s+"="+v;
}

//we store these names as the content of the editor
wo.value(fonts);

//the next button
b.create(10,10,40,30,FL_Regular,FL_NORMAL_BUTTON,"Next");
w.end();
w.resizable(wo);

w.run();
```

Line shapes

Tamgultk provides the following values as line shapes:

FL_SOLID;
FL_DASH;
FL_DOT
FL_DASHDOT
FL_DASHDOTDOT
FL_CAP_FLAT
FL_CAP_ROUND
FL_CAP_SQUARE
FL_JOIN_MITER
FL_JOIN_ROUND
FL_JOIN_BEVEL

Cursor Shapes

FL_CURSOR_DEFAULT: the default cursor, usually an arrow.
FL_CURSOR_ARROW: *an arrow pointer.*
FL_CURSOR_CROSS: *crosshair.*
FL_CURSOR_WAIT: watch or hourglass.
FL_CURSOR_INSERT: *I-beam.*
FL_CURSOR_HAND: hand (up arrow on MSWindows).
FL_CURSOR_HELP: *question mark.*
FL_CURSOR_MOVE: 4-pointed arrow.
FL_CURSOR_NS: up/down arrow.
FL_CURSOR_WE: left/right arrow.
FL_CURSOR_NWSE: diagonal arrow.
FL_CURSOR_NESW: *diagonal arrow.*
FL_CURSOR_NONE: *invisible.*
FL_CURSOR_N: for back compatibility.
FL_CURSOR_NE: for back compatibility.
FL_CURSOR_E: for back compatibility.
FL_CURSOR_SE: for back compatibility.
FL_CURSOR_S: for back compatibility.
FL_CURSOR_SW: for back compatibility.
FL_CURSOR_W: for back compatibility.
FL_CURSOR_NW: for back compatibility.

Simple window

The philosophy in FLTK is to open a window object, to fill it with as many widgets as you wish and then to close it. Once, the window is ready, you simply *run* it to launch it.

```
use("gui");

//We first declare our window object
window w;
//We then begin our window instanciation
w.begin(300,200,1300,150,"Modification");
//We want our window to be resizable
w.sizerange(10,20,0,0);
//we create our winput, which is placed within the current window
txt.create(200,20,1000,50,true,"Selection");
//no more object, we end the session
w.end();

//we then launch our window
w.run();
```

If we do not want to store any widgets in our window, we can replace a call to *begin* with a final *end*, with *create.*

Drawing window

If you need to draw things, such as lines or circles, then in that case, you must provide the window with a new drawing function.

In Tamgu, this function is provided through a simple *with* keyword, together with the object, which will be passed to the drawing function.

```
window wnd(object) with callback_window;
```

This declaration requires some explanations. First, the "*with"* introduces the new *display* function, which the window will use for its drawings. If a *redraw* is applied to this *window*, then this function will be automatically called. Second, *object* is the variable that will be automatically passed to the associate function, when this function is called.

The associate function must expose the following signature:

```
function callback_window(window w, type o) {…}
```

w is our current window, while *o* is the object, which was declared with the window. This function should be a sequence of drawing methods, as the one described above.

Example

```
use("gui");

//A small frame to record our data
frame mycoord {

    int color;
    int x,y;

    function _initial() {
        color=FL_RED;
        x=10;
        y=10;
    }
}

//we declare our object, which will record our data
mycoord coords;

//our new display...
//Every time the window will be modified, this function will be called with a
mycood object
function display(window w,mycoord o) {
    //we select our color, which will be apply to all objects that follow
    w.drawcolor(o.color);
    //a different line shape
    w.lineshape(FL_DASH,10);
    //we draw a rectangle
    w.rectangle(o.x,o.y,250,250);
    //with some text...
    w.drawtext("TEST",100,100);
}

//we declare our window together with its associated drawing function and the
object coords
window wnd(coords) with display;

//We do not need any widgets
wnd.create(100,100,300,300,"Drawing");
wnd.run();
```

Mouse

It is also possible to track the different mouse actions through a callback function. The method *mouse* has been provided at this effect. It associates a mouse action with a call to a specific callback function:

onmouse(action,callback,myobject);

1) *action* must be one of the following values:

FL_PUSH: when a button has been pushed
FL_RELEASE: when a button has been released
FL_MOVE: when the mouse moves
FL_DRAG: when the mouse is dragged
FL_MOUSEWHEEL: when the mouse wheel is moved

2) The callback function must have the following signature:

function callback_mouse(window w, map coords, type myobject);

The first parameter is the window itself. The second parameter is a map with the following keys:

coords["button"]	the value of the last button that was pushed (1,2 or 3)
coords["x"]	the X coordinate within the window of the mouse
coords["y"]	the Y coordinate within the window of the mouse
coords["xroot"]	the mouse absolute X coordinate
coords["yroot"]	the mouse absolute Y coordinate
coords["wheelx"]	the mouse wheel increment on X
coords["wheely"]	the mouse wheel increment on Y

3) *myobject* is the object that will be passed to the callback function

Example

```
use("gui");

//we declare our object, which will record our data
mycoord coords;

//our new display...
//Every time the window will be modified, this function will be called with a
mycood object
function display(window w,mycoord o) {
    //we select our color, which will be apply to all objects that follow
    w.drawcolor(o.color);
    //a different line shape
    w.lineshape(FL_DASH,10);
    //we draw a rectangle
    w.rectangle(o.x,o.y,250,250);
    //with some text...
    w.drawtext("TEST",100,100);
}

//This function will be called for every single move the mouse, the mycoord object
//is the same as the one that is associated with our window
function move(window w,map mousecoord,mycoord o) {
    //we then use the mouse coordinates to position our rectangle
    o.x=mousecoord["x"];
    o.y=mousecoord["y"];
    //we then redraw our window...
    w.redraw();
}

//we declare our window together with its associated drawing function and the
object coord
window wnd(coords) with display;

//We need to instanciate the mouse callback
wnd.begin(100,100,300,300,"Drawing");
//the window will be resizable
wnd.sizerange(10,10,0,0);
//we add a mouse callback, every MOVE of the mouse will
```

```
//trigger a call to “move”. We share the same object coords with the window
wnd.onmouse(FL_MOVE,move,coords);
//the end...
wnd.end();

wnd.run();
```

Example: Plot a Graph

```
use("gui");

fvector fxy;

//first we compute a graph and we store the values in fxy.
//even positions correspond to x values
//odd positions correspond to y values
function mypoints() {
    float x,y;
    for (x in <-20,20,0.1>) {
        y=x*x*x-10;
        fxy.push(x);
        fxy.push(y);
    }
}

//This function will plot our graph
function graph(window w,self o) {
    w.drawcolor(FL_BLACK);

    //We plot our graph, which returns some values, with the following
    //interpretation: [maxWindowX,maxWindowY,minxValue,minyValue,
    //maxXValue,maxYValue,incrementX,incrementY]
    fvector landmarks=w.plot(fxy,0);

    //We then compute the 0,0 coordinates in this new dimension
    fvector axes=w.plotcoords(landmarks,0,0);

    //We draw the axes
    w.line(axes[0],0,axes[0],landmarks[3]);
    w.line(0,axes[1],landmarks[2],axes[1]);
}


mypoints();
window w with graph;

w.begin(30,30,1000,800,"Graph");
w.sizerange(30,30,2000,2000);
w.end();
w.run();
```

Keyboard

It is also possible to associate a keyboard action with a callback function. The function to be used in this case is:

onkey(action,callback,myobject);

1) *action* must be one of the following values:

FL_KEYUP: when a key is pushed
FL_KEYDOWN: when a key is released

2) The callback function must have the following signature:

function callback_key(window w, string skey,int ikey,int combine, myobject object);

The first parameter is the window itself. The second parameter is the text matching the key that was pressed, the third parameter is the key code, the fourth one is a combination of all command keys that were pressed together with the current key and the last one the object that was provided with the *key function*.

3) *object* is the object that will be passed to the callback function

Combination

The combination value is a binary coded integer with the following possible values:

- 1=the ctrl-key was pressed

- 2=the alt-key was pressed
- 4=the command-key was pressed
- 8=the shift-key was pressed

Example

```
use("gui");

//we declare our window together with its associated drawing function and the object
function pushed(window w,string skey,int ikey,int comb,mycoord o) {
    //If the key which is pushed is "J", then we move our rectangle by 10 pixels up and down
    if (skey=="J") {
        o.x+=10;
        o.y+=10;
        //we redraw the whole stuff, so that the coordinates are
        //taken into account
        w.redraw();
    }
}


window wnd(coords) with display;

//We need to instanciate the mouse callback
wnd.begin(100,100,300,300,"Drawing");
//the window will be resizable
wnd.sizerange(10,10,0,0);
//we add a mouse callback, every MOVE of the mouse will
//trigger a call to "move". We share the same object coords with the window
wnd.onkey(FL_PUSH,pushed,coords);
//the end...
wnd.end();
wnd.run();
```

How to add a menu

Adding a menu to a window requires a little more work than for the other elements of the interface. A menu is composed of a series of top menu items, and for each of these top menu items, you must provide a specific description of the sub-menus. Each sub-menu is also associated with a callback function, whose signature must match the following:

```
function callback_menu(window w,myobj obj);
```

where *obj* is an object provided by the user, within the sub-menu description.

A menu item is described through a vector, where the first element is the menu item name, followed with a series of vectors, where each element is a sub-menu.

```
vector menu;
menu.push(["&File",["&New File",[FL_COMMAND,"o"],cmenu1,obj,true],
    ["&Open File",[FL_COMMAND,"i"],cmenu2,obj,false]]);

menu.push(["&Edit",["Cu&t",[FL_COMMAND,"x"],cmenu4,obj,true],
   ["&Copy",[FL_COMMAND,"c"],cmenu3,obj,false]]);
```

In the example above, we add two menu items, whose name are *File* and *Edit*, with for each two sub-menus.

A sub-menu item comprises the following fields:

a) Its name: "&New File"

b) Then a combination of keys, which might trigger the sub-menu base either on the key code or associated with one of the following values:

FL_SHIFT	*SHIFT*
FL_CAPS_LOCK	*CAPS Lock*
FL_CTRL	CONTROL (see FL_CONTROL)
FL_ALT	*ALT key*
FL_NUM_LOCK	*NUM LOCK*
FL_SCROLL_LOCK	*SCROLL Lock*
FL_COMMAND	COMMAND (see Mac OS)
FL_CONTROL	Equivalent to FL_CTRL

c) The callback function itself

d) The associated object, which is passed to the callback

e) A Boolean value to add a sub-menu separator.

Once this vector has been described, you can use the *menu* method in window to load it: *w.menu(menu,5,5,100,20);*

Moving rectangle

Since FLTK is event-based, animation can be done with a proper function. The only way is to use a thread, which will run on its own, independently from the window environment.

```
use("gui");
//A small frame to record our data
frame mycoord {
    int x,y;

    function _initial() {
        x=0;
        y=0;
    }
}

//we declare our object, which will record our data
mycoord coords;

bool first=true;
//our new display...
//Every time the window will be modified, this function will be called with a
mycood object
function display(window w,mycoord o) {
    if (first) {
        w.drawcolor(FL_RED);
        w.drawtext("Press T",20,20);
    }
    else {
        //we select our color, which will be apply to all objects that follow
        w.cursorstyle(FL_CURSOR_CROSS,FL_BLACK,FL_WHITE);
        w.drawcolor(FL_RED);
        w.rectangle(o.x,o.y,60,60);
        //with some text...
        w.drawtext("TEST",o.x+20,o.y+20);
    }
}
```

```
//Once triggered, this thread will increment the coordinates and forces a redraw of
the window for each new value.
thread bouge(window wnd) {
    while (true) {
        coords.x++;
        coords.y++;
        wnd.redraw();
    }
}


function pressed(window w,string skey,int ikey,int comb,mycoord o) {
    //If you press T then the rectangle will start moving...
    if (skey=="T") {
        first=false;
        bouge(w);
    }
}



//we declare our window together with its associated drawing function and the
object coord
window wnd(coords) with display;

//We need to instanciate the mouse call back
wnd.begin(100,100,1300,900,"Drawing");
wnd.sizerange(10,10,0,0);
//we add a mouse call back
wnd.onkey(FL_KEYUP,pressed,coords);
wnd.end();

wnd.run();
```

Thread: moving balls

```
use("gui");
```

```
int nb=0;
frame mycoord {

    int color;
    int x,y,ix,iy;
    //common means that these values are common to all objects
    common int maxx,maxy;
    int idx;

    function _initial(int xx,int yy) {
        color=FL_RED;
        x=xx;
        y=yy;
        ix=1;
        iy=1;
        idx=nb;
        nb++;
    }

    function Idx() {
        return(idx);
    }

    function increment() {
        x+=ix;
        if (x>=maxx)
            ix=-1;
        else
            if (x<=0)
                ix=1;
        y+=iy;
        if (y>=maxy)
            iy=-1;
        else
            if (y<=0)
                iy=1;
    }

    function raz() {
        x=10;
        y=10;
    }

    function X() {
        return(x);
    }

    function Y() {
        return(y);
    }

    function string() {
        string s=x+","+y;
        return(s);
    }
}


//This thread increments the coordinates of the ball
thread move(window w,mycoord ballecoords) {
    while (true) {
        ballecoords.increment();
        //we redraw our window to take these new coordinates into account
        try {
            w.redraw();
        }
        catch() {
            return;
        }
    }
}

//We create a base object to handle the window size
mycoord basecoords(0,0);
basecoords.maxx=500;
basecoords.maxy=300;
int debx,deby;

vector balles;
```

```
function clicked(button b,window w) {
    //the initial positions of the ball are random
    debx=random()*500;
    deby=random()*300;
    //we create our new ball
    mycoord ballecoords(debx,deby);
    //we keep a track of these coordinates in a vector
    balles.push(ballecoords);
    move(w,ballecoords);
}

function display(window w,vector bs) {
    //we select our color, which will be apply to all objects that follow
    self o;
    int i;
    //for each ball, we draw a simple circle with its number in the middle
    for (o in bs) {
        w.circle(o.X(),o.Y(),10);
        w.drawtext(o.Idx(),o.X()-5,o.Y()+2);
    }
    //if the dimensions of the windows have changed, we use them as new
constraints...
    basecoords.maxx=w.coords()[2];
    basecoords.maxy=w.coords()[3];
}


//we declare our window together with its associated drawing function and the
object coord
window wnd(balles) with display;
//We need to instanciate the mouse call back
wnd.begin(100,50,500,300,"Drawing");

wnd.sizerange(10,10,0,0);

button b(wnd) with clicked;
b.create(10,10,20,20,FL_Regular,FL_NORMAL_BUTTON,"Ok");


wnd.end();
wnd.run();
```

Creating windows in a thread

It is possible to create windows within a thread but with some specific precautions. FLTK does not allow the creation of windows within a thread per se, however a message mechanism is available which can be used to post window creation or enrichment requests.

First, a lock should be set around the window creation itself to avoid problems.

Second, a timeout should also be defined to avoid any inner locking when the creation of a window fails.

Third, if any problem occurs during the widget creation, then the *window under scope must be closed.*

Finally, whenever a window is moved or modified by a user, this might result into a momentary freeze of other thread display, since the display and update of windows can only be done within the main thread.

Example

```
use("gui");

int px=300;
int py=400;
int nb=1;

//This thread will display a counter
thread bouge() {
    int i=0;
    window wx;
    woutput wo;
    string err;

    //We initialize our main window with a timeout that will be shared by all sub-
objects
    wx.timeout(0.1);
    //Our main lock, so that only one thread can create a window at a time
    lock("creation");
    try {
        wx.begin(px,py,250,100,"ICI:"+nb);
        wo.create(50,20,120,30,true,"Valeur");
        wx.end();
        px+=300;
        nb++;
        if (px>=1800) {
            px=300;
            py+=150;
        }
    }
    catch(err) {
        //Any errors, we stop. VERY IMPORTANT, we close the window
        if (wx.created())
            wx.close();
        unlock("creation");
        return;
    }
    //We release our lock, so other windows can also be created
    unlock("creation");
    //We set a different time out for the counter display
    wo.timeout(1);


    while (true) {
        i++;
        try {
            wo.value(i);
        }
        catch(err) {
            //If it is a time out we carry on
            if ("Time out" in err)
                continue;
            //Else we clean our slate back
            if (wx.created())
                wx.close();
            return;
        }
    }
}
```

```
function pressed(button b,self n) {
    bouge();
}


//we declare our window together with its associated drawing function and the
object coord
window wnd;
//We need to instanciate the mouse call back
wnd.begin(100,50,500,300,"Drawing");
button b with pressed;
b.create(430,20,60,60,FL_Regular,FL_NORMAL_BUTTON,"Ok");
wnd.sizerange(10,10,0,0);
//we add a mouse call back
wnd.end();

wnd.run();
```

52.5 browser (browsing strings)

The *browser* object defines a box in which strings can be displayed and if necessary selected as a list.

Methods

1. **add(label)**: *Add a string to the browser*
2. **clear()**: *Clear the browser from all values*
3. **columnchar()**: *Return the column char separator.*
4. **columnchar(string)**: *Set the column char separator*
5. **create(x,y,w,h,label)**: Create a browser
6. **deselect()**: *Deselect all items*
7. **deselect(int i)**: *Deselect item i;*
8. **formatchar()**: *Return the format char.*
9. **formatchar(string)**: *Set the format char*
10. **insert(l,label)**: *Insert a label before line l*
11. **load(filename)**: *Load a file into the browser*
12. **select()**: *Return selected string.*
13. **select(int i)**: *Return string at position i.*
14. **size()**: *Return the number of values within the browser*

15. **value()**: *return the current selected value as an index*

Selection

The only way to use browser in selection mode is to associate it with a callback function whose signature must match the following:

function browser_callback(browser b,myobject o);

A callback function is declared with “with”.

Example

```
use("gui");

//the callback function
function avec(browser b,self n) {
    println("Selection:",b.select(),b.value());
}

window w;

w.begin(40,40,400,500,"Browsing");

browser b with avec;
b.create(10,10,100,150,"Test");
b.add("first");
b.add("second");
b.add("third");
b.add("fourth");

w.end();
w.run();
```

52.6 wtree and wtreeitem

These two objects are used to handle a tree, which is clickable. The first object is the tree object itself, which is composed of a set of *wtreeitem*.
The object which is displayed is a hierarchy of nodes, which can each be selected through a callback function.

wtree Methods

1. **add(string path)**: *Add a tree item and return the new wtreeitem*
2. **add(wtreeitem e,string n)**: *Add a tree item after e and return the new wtreeitem.*
3. **clear()**: *Delete the tree items*
4. **clicked()**: *Return the element that was clicked.*
5. **close(string path)**: *Close the element.*
6. **close(wtreeitem e)**: *Close the element.*
7. **connectorcolor(int c)**: *Set or return the connector color.*
8. **connectorstyle(int s)**: *Set or return the connector style (see below)*
9. **connectorwidth(int s)**: *Set or return the connector width.*
10. **create(int x,int y,int h,int w,string label)**: *Create a tree*
11. **find(string path)**: *Return the element matching the path.*
12. **first()**: *Return the first element.*
13. **insert(wtreeitem e,string label,int pos)**: *Insert an element after e with label at position pos in the children list.*
14. **insertabove(wtreeitem e,string label)**: *Insert an element above e with label.*
15. **isclosed(string path)**: *Return true if element is closed.*
16. **isclosed(wtreeitem e)**: *Return true if element is closed.*
17. **itembgcolor(int c)**: *Set or return the item background color.*
18. **itemfgcolor(int c)**: *Set or return the foreground color.*
19. **itemfont(int c)**: *Set or return the item font.*
20. **itemsize(int c)**: *Set or return the item font size.*
21. **last()**: *Return the last element as a wtreeitem*
22. **marginleft(int s)**: *Set or Get the amount of white space (in pixels) that should appear between the widget's left border and the tree's contents.*
23. **margintop(int s)**: *Set or Get the amount of white space (in pixels) that should appear between the widget's top border and the top of the tree's contents.*
24. **next(wtreeitem e)**: *Return the next element after e as a wtreeitem.*
25. **open(string path)**: *Open the element.*
26. **open(wtreeitem e)**: *Open the element.*
27. **previous(wtreeitem e)**: *Return the previous element before e as a wtreeitem.*
28. **remove(wtreeitem e)**: *Remove the element e from the tree.*
29. **root()**: *Return the root element as a wtreeitem.*

30. **rootlabel(string r)**: *Set the root label.*
31. **selectmode(int s)**: *Set or return the selection mode (see below)*
32. **sortorder(int s)**: *Set or return the sort order (see below)*

wtreeitem Methods

1. **activate()**: *Activate the current element.*
2. **bgcolor(int c)**: *Set or return the item background color.*
3. **child(int i)**: *Return the child element at position i.*
4. **children()**: *Return number of children.*
5. **clean()**: *Remove the object associated through value.*
6. **deactivate()**: *Deactivate the current element.*
7. **depth()**: *Return the depth of the item.*
8. **fgcolor(int c)**: *Set or return the foreground color.*
9. **font(int c)**: *Set or return the item font.*
10. **fontsize(int c)**: *Set or return the item font size.*
11. **isactive()**: *Return true if element is active.*
12. **isclosed()**: *Return true if element is closed.*
13. **isopen()**: *Return true if element is open.*
14. **isroot()**: *Return true if element is root.*
15. **isselected()**: *Return true if element is selected.*
16. **label()**: *Return the item label.*
17. **next()**: *Return the next element.*
18. **parent()**: *Return the last element.*
19. **previous()**: *Return the previous element.*
20. **value()**: *Return the value associated with the object.*
21. **value(object)**: *Associate the item with a value.*

Callback

It is possible to associate a *wtree* object with a callback. This callback must have the following signature:

```
function wtree_callback(wtree t,wtreeitem i,int reason,myobject o);
```

This function will be called each time an item will be selected from the tree. *Reason* is one of the following values:

```
FL_TREE_REASON_NONE : unknown reason
FL_TREE_REASON_SELECTED : an item was selected
FL_TREE_REASON_DESELECTED :an item was de-selected
FL_TREE_REASON_OPENED : an item was opened
FL_TREE_REASON_CLOSED :an item was closed
```

Iterator

The *wtree* object is iteratable.

Path

Certain functions such as *add* or *find* requires a path. A path is similar to a *unix path* and defines a path from the root to the leaf:

Example: “/Root/Top/subnode”

Connector style

The style of connectors between nodes is controlled by the following flags:

FL_TREE_CONNECTOR_NONE	Use no lines connecting items.
FL_TREE_CONNECTOR_DOTTED	Use dotted lines connecting items (default)
FL_TREE_CONNECTOR_SOLID	Use solid lines connecting items.

Selection mode

The way nodes are selected in the tree is controlled by the following flags:

FL_TREE_SELECT_NONE	selected when items are clicked.
FL_TREE_SELECT_SINGLE	Single item selected when item is clicked (default)
FL_TREE_SELECT_MULTI	items can be selected by clicking with.

Sort order

Items can be added to the tree in an ordered manner controlled with the following flags:

FL_TREE_SORT_NONE	No sorting; items are added in the order defined (default).
FL_TREE_SORT_ASCENDING	Add items in ascending sort order.
FL_TREE_SORT_DESCENDING	Add items in descending sort order.

Example

```
use("gui");
```

```
//this function is called whenever an item is selected or deselected
function avec(wtree t,wtreeitem i,int reason,self n) {
    //we change the size of the selected element
    if (reason==FL_TREE_REASON_SELECTED)
        i.fontsize(20);
    else //the deselected element gets its previous size
        if (reason==FL_TREE_REASON_DESELECTED)
            i.fontsize(FL_NORMAL_SIZE);
}

window w;
wtree mytree with avec;
wtreeitem ei;

w.begin(40,40,400,500,"Browsing");
mytree.create(20,20,150,250,"Tree");

mytree.rootlabel("Root");
ei=mytree.add("Subroot");
mytree.add(ei,"Test");
mytree.add(ei,"Other");
//we add a new element as path
mytree.add("/Subroot/New item");
w.end();

//we modify the font for each element aftward
//This is equivalent to mytree.itemfont(FL_TIMES_BOLD), before adding
elements
iterator it=mytree;
for (it.begin();it.nend();it.next())
    it.value().font(FL_TIMES_BOLD);


w.run();
```

Example from a tree object

```
use("gui");

tree atree={'a':{'b':{'c':1},'d':3}};

function traversetree(tree t,wtree wt,wtreeitem e) {
    if (t==null)
        return;

    wtreeitem x;

    //First element is null
    if (e==null)
        x=wt.add(t);
    else
        x=wt.add(e,t);

    if (t.childnode()!=null)
        traversetree(t.childnode(),wt,x);
    traversetree(t.nextnode(),wt,e);
}


window w;
wtree mytree;

w.begin(40,40,1000,900,"Display tree");
mytree.create(20,20,950,850,"my tree");

//The root of tree becomes the root of its representation
mytree.rootlabel(atree);

//we traverse our tree to build the representation out of it…
traversetree(atree.childnode(),mytree,null);
w.end();
w.run();
```

52.7 winput (input zone)

The *winput* object defines an input area in a window, which can be used in conjunction with a callback function, which will be called when the zone is dismissed.

Methods

1. **i[a]**: *Extract character from the input at position a*
2. **i[a**:*b]: Extract characters between a and b*
3. **color(int c)**: *set or return the text color*
4. **create(int x,int y,int w,int h,boolean multiline,string label)**: *Create an input area with multiline if this parameter is true*
5. **font(string s)**: *set or return the text font*
6. **fontsize(int c)**: *set or return the text font size*
7. **insert(string s,int p)**: *insert s at position p in the input*
8. **selection()**: *return the selected text in the input*
9. **value()|(string v)**: *return the input buffer or set the initial buffer*
10. **word(int pos)**: *return the word at position pos*

Example

```
use("gui");

frame block {
    //We first declare our window object
    window w;
    string final;

    function result(winput txt,block bb) {
        //we store the content of that field in a variable for further use
        final=txt.value();
    }

    //We first declare our winput associated with result
    winput txt(this) with result;

    function launch() {
        //We then begin our window instanciation
        w.begin(300,200,1300,150,"Modification");
        //We want our window to be resizable
        w.sizerange(10,20,0,0);
        //we create our multiline winput, which is placed within the current
        //window
        txt.create(200,20,1000,50,true,"Selection");
        //We initialize our input with some text
        txt.value("Some Input Text");
        //The text will be in BLUE
        txt.color(FL_BLUE);
        //no more object, we end the session
        w.end();
        //we want our text to follow the size of the main window
        w.resizable(txt);
        //we then launch our window
        w.run();
    }
}
//We open a block
block b;
//which will display our input
b.launch();
//b.final contains the string that was keyed in
println("Result:",b.final);
```

52.8 woutput (Output area)

This type is used to create a specific output in a window. It exposes the following methods:

Methods

1. **color(int c)**: *set or return the text color*
2. **create(int x,int y,int w,int h,boolean multiline,string label)**: *Create an output area with multiline if this parameter is true*
3. **font(string s)**: *set or return the text font*
4. **fontsize(int c)**: *set or return the text font size*
5. **value(string v)**: *initialize the buffer*

52.9 box (box definition)

This type is used to draw a box in the main window with some texts. It exposes the following methods:

Methods

1. **create(int x,int y,int w,int h, string label):** *Create a box with a label*
2. **type(int boxtype)***: modify the box type (see below for a list of box types)*

Box types

FL_NO_BOX
FL_FLAT_BOX
FL_UP_BOX
FL_DOWN_BOX
FL_UP_FRAME
FL_DOWN_FRAME
FL_THIN_UP_BOX
FL_THIN_DOWN_BOX
FL_THIN_UP_FRAME
FL_THIN_DOWN_FRAME
FL_ENGRAVED_BOX
FL_EMBOSSED_BOX
FL_ENGRAVED_FRAME
FL_EMBOSSED_FRAME
FL_BORDER_BOX
FL_SHADOW_BOX
FL_BORDER_FRAME
FL_SHADOW_FRAME
FL_ROUNDED_BOX
FL_RSHADOW_BOX
FL_ROUNDED_FRAME
FL_RFLAT_BOX
FL_ROUND_UP_BOX
FL_ROUND_DOWN_BOX
FL_DIAMOND_UP_BOX
FL_DIAMOND_DOWN_BOX
FL_OVAL_BOX
FL_OSHADOW_BOX
FL_OVAL_FRAME
FL_OFLAT_BOX

FL_PLASTIC_UP_BOX
FL_PLASTIC_DOWN_BOX
FL_PLASTIC_UP_FRAME
FL_PLASTIC_DOWN_FRAME
FL_PLASTIC_THIN_UP_BOX
FL_PLASTIC_THIN_DOWN_BOX
FL_PLASTIC_ROUND_UP_BOX
FL_PLASTIC_ROUND_DOWN_BOX
FL_GTK_UP_BOX
FL_GTK_DOWN_BOX
FL_GTK_UP_FRAME
FL_GTK_DOWN_FRAME
FL_GTK_THIN_UP_BOX
FL_GTK_THIN_DOWN_BOX
FL_GTK_THIN_UP_FRAME
FL_GTK_THIN_DOWN_FRAME
FL_GTK_ROUND_UP_BOX
FL_GTK_ROUND_DOWN_BOX
FL_FREE_BOXTYPE

52.10 button

The button object is of course very important as it allows users to communicate with the GUI. A button must be created in connection with a callback whose signature is the following:

```
function callback_button(button b, myobj obj) {…}

button b(obj) with callback_button;
```

It exposes the following methods:

Methods

1. **align(int)**: *define the button label alignment*
2. **bitmap(bitmap im,int color,string label,int labelalign)**: *Use the bitmap as a button image*
3. **color(int code)**: *Set the color of the button*
4. **create(int x,int y,int w,int h,string type,int shape,string label)**: *Create a button, see below for a list of types and shapes*
5. **image(image im,string label,int labelalign)**: *Use the image as a button image*

6. **shortcut(string keycode)**: *Set a shortcut to activate the button from the keyboard (see below for a list of shortcuts code)*
7. **value()**: *return the value of the current button*
8. **when(int when1, int when2,...)**: *Type of event for a button which triggers the callback (see events below)*

Button types

FL_Check
FL_Light
FL_Repeat
FL_Return
FL_Round
FL_Regular
FL_Image

Button shapes

FL_NORMAL_BUTTON
FL_TOGGLE_BUTTON
FL_RADIO_BUTTON
FL_HIDDEN_BUTTON

Events (when)

Below is a list of events, which can be associated with the callback function.
FL_WHEN_NEVER
FL_WHEN_CHANGED
FL_WHEN_RELEASE
FL_WHEN_RELEASE_ALWAYS
FL_WHEN_ENTER_KEY
FL_WHEN_ENTER_KEY_ALWAYS

Shortcuts

Below is the list of shortcuts that can be associated with a button:

FL_Button
FL_BackSpace
FL_Tab
FL_Enter
FL_Pause
FL_Scroll_Lock
FL_Escape

FL_Home
FL_Left
FL_Up
FL_Right
FL_Down
FL_Page_Up
FL_Page_Down
FL_End
FL_Print
FL_Insert
FL_Menu
FL_Help
FL_Num_Lock
FL_KP
FL_KP_Enter
FL_KP_Last
FL_F_Last
FL_Shift_L
FL_Shift_R
FL_Control_L
FL_Control_R
FL_Caps_Lock
FL_Meta_L
FL_Meta_R
FL_Alt_L
FL_Alt_R
FL_Delete
FL_Delete

Example

```
use("gui");

frame block {
  //We first declare our window object
  window w;
  winput txt;
  string final;

  //When the button is pressed, this function is called
  function gettext(button b,block bb) {
    final=txt.value();
```

```
        w.close();
    }


    function launch(string ph) {
        final=ph;
        //We then begin our window instanciation
        w.begin(300,200,1300,150,"Modification");
        //We want our window to be resizable
        w.sizerange(10,20,0,0);
        //we create our winput, which is placed within the current window
        txt.create(200,20,1000,50,true,"Selection");
        txt.value(ph);
        //We associate our button with the method gettext
        button b(this) with gettext;
        b.create(1230,20,30,30,FL_Regular,FL_NORMAL_BUTTON ,"Ok");
        //no more object, we end the session
        w.end();
        w.resizable(txt);
        //we then launch our window
        w.run();
    }
}

block b;
b.launch("My sentence");
```

Image

First, we need to load an image, then we create a button with the flag: FL_Image.

```
image myimage;
//We load a GIF image
image.loadgif('c:\...');
//We associate our button with the method gettext
button b(this) with gettext;
//We create pour image button
b.create(1230,20,30,30,FL_Image,FL_NORMAL_BUTTON ,"Ok");
//which we associate with our button, with an inside label within the image…
```

```
b.image(myimage,"Inside", FL_ALIGN_CENTER);
```

52.11 wchoice

Tamgultk provides a specific type to propose selections in list. This element must be initialized with a specific menu, which we will describe later on.

It exposes the following methods.

Methods

1. **create(int x,int y,int w,int h,string label)**: *Create an choice*
2. **font(string s)**: *set or return the text font*
3. **fontsize(int c)**: *set or return the text font size*
4. **menu(vector s)**: *Initialize the menu. This should be the last operation in a wchoice creation.*
5. **value(int s)**: *set the choice initialization value*

Menu

A menu description is a vector of vectors, each containing three elements.

vmenu=[["First",callback,"1"],["Second",callback,"2"],["Third",callback,"3"]];

A menu item contains, first its name, then the callback function it is associated with then the object that will be passed to this callback function.

Menu Item: [name,callback,object]

The callback function must have the following signature:

function callback_menu(wchoice c, myobject obj);

This function is called for each selection from the list.

Example

```
use("gui");

window w;

function callback_menu(wchoice c, string s) {
    println(s);
}

vector vmenu;
//Our menu description
vmenu=[["Premier",callback_menu,"RRRR"],["second",callback_menu,"OOOOOO
"],["third",callback_menu,"BBBBBBB"]];

wchoice wch;

//we create our window
w.begin(300,200,1300,500,"Fenetre");
//we create our choice widget
wch.create(20,420,100,50,"Choix");
wch.fontsize(20);
//This should be the last operation on the selection list…
wch.menu(vmenu);
w.end();
w.run();
```

52.12 wtable

Tamgultk provides a specific type to display values in a table and select some elements. This element table must be created with a callback function (as most widgets), whose signature is the following:

```
function callback_table(table x,map values,myobject obj);

  table t(obj) with callback_table;
```

The *values* is a map, which contains the following keys:

“top”:	the top row
“bottom”:	the bottom row
“left”:	the left column
“right”:	the right column
“values”:	a map, whose key is a string: “r:c”, with r as row and c as the column.

This object exposes the following methods:

Methods

1. **add(int R,int C,string v)**: *Add a value on row R and column C. The size of the table depends on the number of values added.*
2. **boxtype(int boxtype)**: *box type*
3. **cell(int R,int C)**: *Return the value at row R and column C*
4. **cellalign(align)**: *Set the alignment in a cell*
5. **cellalignheaderrow(align)**: *Set the alignment in a row header cell*
6. **cellalignheadercol(align)**: *Set the alignment in a column header cell*
7. **clear()**: *Clear the table*
8. **colorbg(int c)**: *set or return the cell color background*
9. **colorfg(int c)**: *set or return the cell color foreground*
10. **column()**: *return the number of columns*
11. **column(int nb)**: *Define the number of columns*
12. **columnheader(int C,string label)**: *set the label of the column header for column C*
13. **columnheaderwidth(int sz)**: *the size in pixel of the column header*
14. **columnwidth(int width)**: *Define the column width in pixel*
15. **create(int x,int y,int w,int h,string label)**: *Create a table of objects, and starts adding*
16. **font(int s)**: *set or return the text font*
17. **fontsize(int c)**: *set or return the text font size*
18. **row()**: *return the number of rows*
19. **row(int nb,int height)**: *Define the number of rows*
20. **rowheader(int R,string label)**: *set the label of the row header for row R.*
21. **rowheaderheight(int sz)**: *the size in pixel of the row header*
22. **rowheight(int height)**: *Define the row height in pixel*
23. **selectioncolor(int color)**: *Color for the selected elements*
24. **when(string when)**: *Type of event to trigger the callback*

Example

```
use("gui");
window w;

function callback_table(wtable x,map V,window w) {
   println(V);
}

wtable t(w) with callback_table;
int i,j;
```

```
//we create our window
w.begin(300,200,1300,500,"Fenetre");
//we create our table
t.create(20,20,500,400,"table");
//with a certain font size
t.fontsize(12);
//the selected element will be in blue
t.selectioncolor(FL_BLUE);
//we populate our table
for (i=0;i<10;i++) {
   //including the headers
   t.rowheader(i,"R"+i);
   t.columnheader(i,"C"+i);
   for (j=0;j<10;j++)
      //we populate our table with string of the form: R0C9
   t.add(i,j,"R"+i+"C"+j);
}
//we define the size of rows, with their height in pixels,
//after we populated the table
t.rowheight(20);
//we define the size of columns, with their width in pixels
t.columnwidth(60);
w.end();
w.run();
```

52.13 editor

Tamgultk provides also a specific type to provide users with an editor, which can be used to handle text.

A callback can be associated with an editor, which has a distinctive set of arguments. This callback is triggered whenever the inside text is modified.

function editorcallback(editor e,int pos, int ninserted,int ndeleted,int restyled,string del,myobj obj);

This function is associated with an editor object through a *with* instruction.

editor e(obj) with editorcallback;

The arguments are the following:

editor e: the editor itself
pos: the current cursor position in the document
ninserted: the number of characters which have been inserted
ndeleted: the number of deleted characters
rstyled: the number of characters whose style has been modified
del: the characters which have been deleted
obj: the object which has been associated in the with instruction.

This method exposes the following methods:

Methods

1. **addstyle(map styles)***: Initialize the styles for text chunks (see below for more details)*
2. **annotate(string s,string keystyle,bool matchcase)***: Each occurrence of s in the text is assigned the style keystyle. matchcase is optional.* s *can be a treg. In that case, the matchcase parameter will not be taken into account.*
3. **append(string s)***: append a string at the end of the editor text*
4. **byteposition(int pos)***: convert a character position into a byte position (especially useful in UTF8 strings)*
5. **charposition(int pos)***: convert a byte position into a character position (especially useful in UTF8 strings)*
6. **color(int c)***: set or return the text color*
7. **copy()***: copy selected text to clipboard*
8. **copy(string s)***: copy string s to clipboard*
9. **count(string s,int bg)***: count the number of occurrences of s starting at bg.*
10. **create(int x,int y,int w,int h,string label)***: Create an editor*
11. **cursor()***:return the current position of the cursor in byte increments.*
12. **cursor(int i)***: move the cursor to the* i^{th} *bytes.*
13. **cursorchar()***:return the current position of the cursor in character increments.*

14. **cursorstyle(int style)***: set the cursor shape (see below)*

15. **cut()***: cut selected text to clipboard*

16. **delete()***: delete selected text*

17. **e[a***:b]: Extract characters between a and b*

18. **e[a]***: Extract character from the editor at position a*

19. **find(string s,int i)***: find a string in the editor text, starting at position i.*

20. **font(string s)***: set or return the text font*

21. **fontsize(int c)***: set or return the text font size*

22. **getstyle(int start,int end)***: Return the style for each character of a chunk of text as a vector.*

23. **gotoline(int l,bool highlight)***: goto line l and highlight it if true.*

24. **highlight()***: return 1 or 0 if there is highlighted text in the editor. In the context of a string, returns the highlighted string*

25. **highlight(int start,int end)***: highlight the characters between start and end.*

26. **insert(string s,int pos)***: insert a string at position pos*

27. **line()***: return the current line number or the line text itself*

28. **line(int pos)***: Return the line corresponding to pos or the line text itself. This line should be visible.*

29. **linebounds()***: return a vector with the start position and end position of the current line in bytes increments.*

30. **linebounds(int pos)***: return a vector with the start position and end position of the line at pos in bytes increments.*

31. **lineboundschar()***: return a vector with the start position and end position of the current line in character increments.*

32. **lineboundschar(int pos)***: return a vector with the start position and end position of the line at pos in bytes increments.*

33. **load(string filename)***: load the content of a file into the editor.*

34. **onhscroll(function f, object o)**: *set the callback when scrolling horizontally (see below for an example)*

35. **onkey(int action,function f, object o)**: *set the callback when scrolling vertically (see below for an example)*

36. **onmouse(int action,function f, object o)**: *set the callback when handling the mouse*

37. **onvscroll(function f, object o)**: *set the callback when scrolling vertically (see below for an example)*

38. **paste()**: *paste selected text to clipboard*

39. **rfind(string s,int i)**: *find a string in the editor text, starting at position I, backward.*

40. **save(string filename)**: *save the content of the editor into a file.*

41. **selection()**: *return the selected text in the editor*

42. **setstyle(int start,int end, string keystyle)**: *Set a text chunk with a given style from the style table instatiated with addstyle. (see below for more details)*

43. **unhighlight()**: *remove highlighting*

44. **value(string v)**: *return the text in the editor or initialize the editor*

45. **word(int pos)**: *return the word at position pos*

46. **wrap(boolean w)**: *wrap the text within the editor window if w is true.*

Cursor shape

Tamgu provides different cursor styles:

FL_NORMAL_CURSOR
FL_CARET_CURSOR
FL_DIM_CURSOR
FL_BLOCK_CURSOR
FL_HEAVY_CURSOR

Use *cursorstyle* to set it to the proper value.

Adding styles

In the editor, it is possible to display specific sections of a text with a specific set of fonts, colors and size. However, in order to achieve this display, FLTK requires the description of these styles beforehand. Each item is a vector of three elements: *[color, font, size]* associated with a key, which will be used to refer to that style item.

```
//A map describing the styles available within the editor
map m={'#': [FL_BLACK,FL_COURIER,FL_NORMAL_SIZE ],
  'A': [ FL_BLUE,FL_COURIER_BOLD,FL_NORMAL_SIZE ]
  'B': [ FL_DARK_GREEN,FL_COURIER_ITALIC,FL_NORMAL_SIZE ],
    'C': [ FL_DARK_GREEN, FL_COURIER_ITALIC,FL_NORMAL_SIZE ],
    'D': [ FL_BLUE,FL_COURIER,FL_NORMAL_SIZE ],
    'E': [ FL_DARK_RED,FL_COURIER,FL_NORMAL_SIZE ],
    'F': [ FL_DARK_RED,FL_COURIER_BOLD,FL_NORMAL_SIZE ],
};
```

52.13.1.1.1.1 Important: key “#”

The map should always have a “#” key which is used to define the default style. If this key is not provided, an exception will be raised.

Once this map has been designed, you should pass it to the system with the instruction: *addstyle(m).*

To use this style on a section of text, use *setstyle* with one the above keys as a way to select the correct style.

Example

```
use("gui");
//A map describing the styles available within the editor
map m={'#': [FL_BLACK,FL_COURIER,FL_NORMAL_SIZE ],
'A': [ FL_BLUE,FL_COURIER_BOLD,FL_NORMAL_SIZE ]};
'B': [ FL_DARK_GREEN,FL_COURIER_ITALIC,FL_NORMAL_SIZE ],
  'C': [ FL_DARK_GREEN, FL_COURIER_ITALIC,FL_NORMAL_SIZE ],
  'D': [ FL_BLUE,FL_COURIER,FL_NORMAL_SIZE ],
  'E': [ FL_DARK_RED,FL_COURIER,FL_NORMAL_SIZE ],
  'F': [ FL_DARK_RED,FL_COURIER_BOLD,FL_NORMAL_SIZE ],
window w;
editor e;

w.begin(300,200,1300,700,"Modification");
w.sizerange(10,20,0,0);

e.create(200,220,1000,200,"Editor");
e.addstyle(m);

e.value("This is an interesting style");
//We use the style of key C on interesting
e.setstyle(10,22,'C');
e.annotate("a", 'E'); //each a is assigned the E style
w.end();
w.run();
```

Modifying style

It is actually possible to redefine a style for a given editor. The function *addstyle* must be called again.

```
//A map describing the styles available within the editor
map m={'#':[FL_BLACK,FL_COURIER,FL_NORMAL_SIZE ],
'truc':[ FL_DARK_RED,FL_COURIER,FL_NORMAL_SIZE ]};

//we modify one item in our map... We keep the same key…
//The section in the text based on 'truc' will be all modified…
function test(button b, editor e) {
   m["truc"]=[ FL_DARK_GREEN,FL_COURIER,FL_NORMAL_SIZE ];
   e.addstyle(m);
}

window w;
editor e;
button b(e) with test;

w.begin(300,200,1300,700,"Modification");
w.sizerange(10,20,0,0);

e.create(200,220,1000,200,"Editor");
```

```
e.addstyle(m);
e.value("This is an interesting style");
e.setstyle(10,22,'truc');

b.create(1230,20,30,30,FL_Regular,FL_NORMAL_BUTTON,"Ok");

w.end();
w.run();
```

Style Messages

It is also possible to associate a style with a specific message. This message will be displayed when the mouse will hover above an element having that style. The only modification necessary is to add one or two more elements to each item from the style description.

A style description is composed of: **[itemcolor,font,fontsize].**

We can add a message to that item: [itemcolor,font,fontsize,"Message"].

And even a color which will be used as a background color for that message:

[itemcolor,font,fontsize,"Message",backgroundcolor].

If the background color is not provided, then the defined color *itemcolor* from the style will be used.

Example

```
use("gui");

map m={'#':[FL_BLACK,FL_COURIER,FL_NORMAL_SIZE ],
   'truc':[ FL_DARK_RED,FL_COURIER,FL_NORMAL_SIZE,
"THIS IS A TRUC",FL_YELLOW]};
```

When the mouse will hover above a piece of text with the style *truc*, it will display a yellow box with the message: *THIS IS A TRUC.*

Callbacks: scrolling, mouse and keyboard

The callback must have the following signature

Scrolling callback

function vhscroll(editor e, object n);

Mouse callback

function mouse_callback(editor e, map coords, object n);

The second parameter is a map with the following keys:

coords[“button”]	the value of the last button that was pushed (1,2 or 3)
coords[“x”]	the X coordinate within the window of the mouse
coords[“y”]	the Y coordinate within the window of the mouse
coords[“xroot”]	the mouse absolute X coordinate
coords[“yroot”]	the mouse absolute Y coordinate
coords[“wheelx”]	the mouse wheel increment on X
coords[“wheely”]	the mouse wheel increment on Y
coords[“cursor”]	the mouse cursor position within the editor as a character position

Keyboard callback

function key(editor e, string k, int ikey object n);

In this example, we set three different callbacks with the vertical scrolling, the mouse and the keyboard. Each manipulation will update the line number in an output field.

```
function cvscroll(editor e,woutput num) {
   num.value(e.line());
}

function cmouse(editor e,map coords,woutput num) {
   num.value(e.line());
}

function ckey(editor e, string k, int i,woutput num) {
   num.value(e.line());
}
```

```
window w;
editor e;
woutput num;

w.begin(300,200,1300,700,"Window");
w.sizerange(10,20,0,0);
num.create(100,100,30,40,”Line”);

e.create(200,220,1000,200,"Editor");
e.onmouse(FL_RELEASE,cmouse,num);
e.onvscroll(cvscroll,num);
e.onkey(FL_KEYUP,ckey,num);

w.end();
w.run();
```

Sticky notes

The following example shows how to display on words with a specific style in your editor a little sticky note.

```
//A map describing the styles available within the editor
map m={'#':[FL_BLACK,FL_COURIER,FL_NORMAL_SIZE ],
'movement':[ FL_RED,FL_COURIER,FL_NORMAL_SIZE ]};

//We define the words that we want to recognize in the text
vector mvt=["move","run","stride","walk","drive"];

//whenever the text is modified, we check for our above words
function modified(editor e,int pos, int ninserted,int ndeleted,int restyled,string
del,self obj) {
    //we unmark everything first
    e.setstyle(0,e.size(),"#");
    //then, we mark all our movement words
    e.annotate(mvt,"movement");
}

//we need our message window to be displayed at the precise location of our
mouse
```

```
window wmessage;
//This method is called whenever the mouse cursor is on a non default style
function infostyle(int x,int y,map sz,string style) {
    //If there is already a sticky note we do nothing
    if (wmessage!=null)
        return;

    //we create a borderless window, with a yellow background
    //sz contains the string dimension in pixels
    wmessage.begin(x,y,sz["w"]+20,sz["h"]+20,style);
    wmessage.backgroundcolor(FL_YELLOW);
    wmessage.border(false);
    box b;
    //we display our style, which is a string
    b.create(5,5,sz["w"]+5,sz["h"]+5,style);
    wmessage.end();

}


//This function is called when the mouse is moved in the editor
function vmouse(editor e,map infos,self n) {
    //we get the style at the cursor position which matches a position in the text
    string style=e.getstyle(infos["cursor"],infos["cursor"]+1);
    //If it is not a standard style
    if (style!='#' && style!="")
        //we create our sticky note
    infostyle(infos["xroot"],infos["yroot"],e.textsize(style),style);
    else
        //if it is the standard style or no style at all, we close our window...
    if (wmessage!=null)
        wmessage.close();
}

window w;
editor e with modified;
//we create our window
w.begin(300,200,1300,700,"Marking movement words");
w.sizerange(10,20,0,0);
//then our editor
```

```
e.create(200,220,1000,200,"Editor");
//we add the style
e.addstyle(m);
//and we also need a mouse callback
e.onmouse(vmouse,null);

w.end();
w.run();
```

52.14 scroll

It is possible to define a scrolling region within a window. The type scroll can be used at this effect.
It exposes the following methods:

Methods

1. **create(int x,int y,int w,int h,string label)**: *Create a scrolling region*
2. **resize(object)**: *make the object resizable*

52.15 wprogress

Tamgultk offers a progress bar widget, which can display a progression from a minimum to a maximum value.
A *wprogress* can be attached to a callback function in order to catch the value modifications.

The function must have the following signature:

```
function callback_progress(wprogress s,myobj obj) {
   //the progress value is returned with value()
   println(s.value());
}

progress s(obj) with callback_progress;
```

The progress object exposes the following functions:

Methods

1. **backgroundcolor(int color)**: *set or return the background color*
2. **barcolor(string code|int code)**: *Set the bar color*
3. **create(int x,int y,int w,int h,int alignment, string label)**: *Create a progress bar*
4. **minimum(float x)**: *defines or return the progress bar minimum*
5. **maximum(float x)**: *defines or return the progress bar maximum*
6. **value(float)**: *define the value for the progress or return its value*

Example:

```
use("gui");

window w;

wprogress c;

thread progressing () {
    for (int i in <0,100,1>) {
        for (int j in <0,100000,1>) {}
        c.value(i);
    }
    printlnerr("End");
}


function launch(button b,self e) {
    progressing();
}

button b with lance;

w.begin(50,50,500,500,"test");
c.create(30,30,300,30,"progression");
b.create(100,100,50,50,"Ok");
```

```
c.minimum(0);
c.maximum(100);
c.barcolor(FL_BLUE);
w.end();
w.run();
```

52.16 wcounter

Tamgultk offers two sorts of counters. One which displays two steps of progression, the other which displays only one.
A *wcounter* must be attached with a callback function in order to catch the value modifications.

The function must have the following signature:

```
function callback_counter(wcounter s,myobj obj) {
  //the counter value is returned with value()
  println(s.value());
}

counter s(obj) with callback_counter;
```

The counter object exposes the following functions:

Methods

1. **bounds(float x,float y)**: *defines the counter boundary*
2. **create(int x,int y,int w,int h,int alignment, string label)**: *Create a counter*
3. **font(int s)**: *set or return the text font*
4. **lstep(float)**: *define the large counter step*
5. **step(float)**: *define the counter step*
6. **steps(float)**: *define the counter steps, normal and large.*
7. **textcolor(string code|int code)**: *Set the color of the text*
8. ***textsize(string l)***: Return a map with w and h as key to denote width and height of the string in pixels
9. ***type(bool normal)***: if 'true' then normal counter or simple counter
10. ***value(float)***: define the value for the counter or return its value

Example:

```
use("gui");

window w;

function tst(wcounter e,self i) {
    printlnerr(e.value());
}

wcounter c with tst;

w.begin(50,50,500,500,"test");
c.create(30,30,300,100,"Counter");
c.steps(0.01,0.1);
c.textsize(20);
c.textcolor(FL_RED);
w.end();
w.run();
```

52.17 slider

Tamgultk offers two sorts of slider. One of these sliders displays a value with the slide bar itself.

The slider must be attached with a callback function in order to catch any modifications. The function must have the following signature:

```
function callback_slider(slider s,myobj obj) {
  //the slider value is returned with value()
  println(s.value());
}

slider s(obj) with callback_slider;
```

The slider exposes the following methods:

Methods

1. **align(int align)**: *define the slider alignement*
2. **bounds(int x,int y)**: *defines the slider boundaries*

3. **create(int x,int y,int w,int h,int align,bool valueslider,string label)**: *Create a slider or a valueslider (see below for a list of alignment values)*
4. **resize(object)**: *make the object resizable*
5. **step(int)**: *define the slider step*
6. **type(int x)**: *Value slider type (see below for the list of slider types)*
7. **value()**:*return the slider value*
8. **value(int i)**: *define the initial value for the slider*

Slider types

FL_VERT_SLIDER
FL_HOR_SLIDER
FL_VERT_FILL_SLIDER
FL_HOR_FILL_SLIDER
FL_VERT_NICE_SLIDER
FL_HOR_NICE_SLIDER

Example

This example shows how a slider can control the movement of a rectangle in another window.

```
use("gui");

//A small frame to record our data
frame mycoord {

    int color;
    int x,y;

    function _initial() {
        color=FL_RED;
        x=0;
        y=0;
    }
}
```

```
//we declare our object, which will record our data
mycoord coords;
//we declare our window together with its associated drawing function and the
object coord
window wnd(coords) with display;

//We cheat a little bit as we use the global variable wnd to
//access our window…
function slidercall(slider s,mycoord o) {
   //we position our window X according to the slider value
   o.x=s.value();
   wnd.redraw();
}

slider vs(coords) with slidercall;

//We need to instanciate the mouse callback
wnd.begin(100,100,300,300,"Drawing");
wnd.sizerange(10,10,0,0);
//we create our value slider
vs.create(10,10,180,20,FL_ALIGN_LEFT,true,"Position");
//the values will be between 0 and 300
vs.bounds(0,300);
//with the initial value 100
vs.value(100);

wnd.end();

wnd.run();
```

52.18 tabs and group

The object *tabs* exposes everything that is necessary to create tabs in a window. This object is associated with the object *group,* which is used to group widgets together in a single block.

Tabs methods

The *tabs* object exposes the following methods:

1. **add(wgroup g)***: dynamically add a new tab.*
2. **begin(int x,int y,int w, int h,string title)***: Create a tab window and begin initialization*
3. **current()***: return the current active tab*
4. **current(wgroup t)***: activate this tab*
5. **end()***: end the tabs construction*
6. **remove(wgroup g)***: remove the group g from the tabs*

Group methods

The *group* object exposes the following methods:

1. **activate()***: activate the tab*
2. **begin(int x,int y,int w, int h,string title)***: Create a widget group and begin initialization*
3. **end()***: end the group construction*

Important

- The creation of a *tabs* section is quite simple. You create a *tabs* box, in which all the different elements will be stowed.
- For each tab, you need to create a specific widget group.
- The dimension of a group should be inferior in height to the *original tabs box*.
- Each group should have the same dimension.
- The second group should always be hidden.

Call back

It is also possible to associate a group with a callback as with *window.* When a group is declared with an associate callback, this callback is called each time the window must be redrawn. See *window* for more information. Most of the functions available for drawing in the object window are also available for *wgroup*.

Simple example

In this example, we build a simple window with two tabs.

```
use("gui");
window wnd;
//We create our main window
wnd.begin(100,100,500,500,"TABS");
//then a tab section
wtabs tabs;
//which we define as a box
tabs.begin(10,55,300,325,"Onglets");

//the first group is a list of widget
wgroup g1;
//we begin loading our widget
//The size is 80=55+25 and the height is 300=325-25
g1.begin(10,80,300,300,"Label&1");
//there will be only one
winput i1;
i1.create(60,90,240,40,true,"Input 1");
//our group 1 is now finished
g1.end();

//then we create our second tab section as a group again
wgroup g2;
//the size of this group is exactly the same as g1
g2.begin(10,80,300,300,"Label&2");
//IMPORTANT: we hide it
g2.hide();
//We add our new widgets
winput i2;
i2.create(60,90,240,40,true,"Input 2");
//our group 2 is now finished
g2.end();
//so are our tabs
tabs.end();

wnd.end();
wnd.run();
```

A more complex example

In this new example, we show how tabs can be dynamically added to an existing tab window. We implement a button, which when pressed, triggers the creation of a new tab. A second button shows how a tab can be removed from the list of tabs.

```
use("gui");
int nb=0;

//This function will delete the current active tab
function removetab(button b, wtabs t) {
    //we get the current active tab in a self since we do not want to have
    //a copy of that element but the actual pointer to this element
    self x=t.current();
    t.remove(x);
}

//this function creates a new tab which is appended to the existing tab structure
function addtab(button b,wtabs x) {
    wgroup g;
    //same size fits all
    g.begin(10,80,300,300,"Label&"+nb);
    //IMPORTANT: we hide it unless it is the first one
    if (nb!=0)
        g.hide();
    //We add our new widgets
    winput i;
    i.create(60,90,240,40,true,"Input "+nb);
    //our group  is now finished
    g.end();
    nb++;
    //we add it to our existing tab structure...
    x.add(g);
}


window wnd;
//We create our main window
wnd.begin(100,100,500,500,"TABS");
//then a tab section
```

```
wtabs tabs;
//which we define as a box
tabs.begin(10,55,300,325,"Onglets");
//The tabs section is of course empty
tabs.end();

//we add a button to trigger the creation of a new tab...
button b(tabs) with addtab;
b.create(400,100,50,30,FL_Regular,FL_NORMAL_BUTTON,"Add");

//This button will delete the current tab
button br(tabs) with removetab;
br.create(400,140,50,30,FL_Regular,FL_NORMAL_BUTTON,"Remove");

wnd.end();
wnd.run();
```

Callback example

```
use("gui");
//Our redrawn function
function drawing(wgroup w,self n) {
    w.drawcolor(FL_BLACK);
    w.circle(100,100,100);
}

//we create a group that is linked with a redrawn function
wgroup fen with drawing;
fen.begin(10,80,300,300,"Infos");
fen.end();
//We then associate it as a tab
tabs.add(fen);
```

52.19 filebrowser

This object is used to display a window to browse your disks to fetch a file or a directory. The object can be created with a callback function, whose signature is the following:

function callback_filebrowser(filebrowser f,myobject object);

However, if you do not declare any callback function, the function *create* returns the selected file pathname.

It exposes the following methods.

Methods

1. **close()**: *Close the file browser*
2. **create(string intialdirectory,string filter,int method,string label)**: *Open a file browser, according to method (see below). If no callback function is declared, then it returns the selected pathname.*
3. **ok()**: *return true if ok was pressed*
4. **value()**: *Return the selected file*

Method

There are different ways to open a filebrowser, each with a different action. The possible values are the following:

- FL_DIR_SINGLE: to open a single file at a time
- Fl_DIR_MULTI: to open multiple files at a time
- FL_DIR_CREATE: to create a new file
- FL_DIR_DIRECTORY: to select a directory

Example

```
use("gui");

//we check wether the element was chosen
//then we close our window
function choose(filebrowser f,self b) {
   if (f.ok()) {
      println("Ok:",f.value());
      b=true;
      f.close();
   }
}
```

```
bool b=false;
filebrowser fb(b) with choose;
fb.create('C:\TAMGU',"*", FL_DIR_SINGLE,"Choose your file");
```

A simpler solution is:

```
filebrowser f;
string value=f.create(".","*.*",FL_DIR_SINGLE,"Open");
println("Value:",value);
```

53 Library sound: type sound

Tamgu also provides a way to play any types of sound files (WAV, MP3, FLAC, OGG etc.).
You simply load a file and you can play it anywhere in your code. It provides at this effect the type: "sound".

N.B. Tamgu relies on *libao4, libsndfile-1* and *libmpg123* for decoding and playing.

To call this library: use("sound")

53.1 Methods

The API exposes the following methods:

1. **close()**: *close a sound channel*
2. **decode(ivector soundbuffer)**: *decode the sound file and returns the content buffer by buffer into soundbuffer. Return false when the end of the file is reached.*
3. **encode(ivector soundbuffer)**: *play a soundbuffer returned by decode.*
4. **load(string pathname)**: *Load the sound pathname.*
5. **open(map params)**: *open a sound channel with the parameters of the current sound file (see parameters)*
6. **parameters()**: *return the parameters of the current sound file as a map.*
7. **parameters(map modifs)**: *Only "rate" and "channels" can be modified.*
8. **play()**: *play the sound.*
9. **play(bool beg)**: *play the sound from the beginning*
10. **play(ivector soundbuffer)**: *play the sound buffer (see encode)*
11. **reset()**: *reset the sound file to the beginning.*

12. **stop()**: *stop the sound. It is necessary to play the sound file in a thread in order to use this instruction.*

Example

```
use("sound");
sound s;

s.load('C:\TAMGU\TAMGU7\sound\Kalimba.mp3');
s.play();
```

You can also load a sound with the following declaration:

sound s('C:\TAMGU\TAMGU7\sound\Kalimba.mp3');

Decoding example

```
use("sound");

//we open a sound file
sound s('C:\TAMGU\TAMGU7\sound\Kalimba.mp3');

//we open a second sound channel
sound c;

//we get the sound parameters
map params=s.parameters();

//which we use to open a channel
c.open(params);

//we loop with decode in the sound file
//and for each new buffer, we play our sound
//we could use "play" instead of "encode",
//but it is little bit slower

ivector snd;
while (s.decode(snd))
   c.encode(snd);
```

```
//then we close our channel...
c.close();
```

54 Library curl: type curl (WEB)

The *curl* type is used to load HTML page from the internet. It is based on the cURL (http://curl.haxx.se/) library and offers some basic tools to handle HTML pages.

The name of the library is *tamgucurl:* use("tamgucurl");

54.1 Methods

1. **execute()**: *to execute a curl query. Options should have been provided.*
2. **execute(string filename)**: *to execute a curl query. Options should have been provided. When filename is supplied, then the output is stored in a file.*
3. **options(string option,string|int parameter)**: *to supply options to curl before either calling execute or url. See below for a list of all available options.*
4. **password(string user,string psswrd)**: *to provide a site with a user and a password*
5. **proxy(string proxy)**: *to set a proxy connection*
6. **url(string uri)**: *to load a url. This command executes a options("CURLOPT_URL",uri) before executing the command itself.*
7. **url(string uri,string filename)**: *to load a url and store the result in a file.*

54.2 Options

CURLOPT_ACCEPTTIMEOUT_MS, CURLOPT_ACCEPT_ENCODING,
CURLOPT_ADDRESS_SCOPE, CURLOPT_APPEND,
CURLOPT_AUTOREFERER, CURLOPT_BUFFERSIZE, CURLOPT_CAINFO,
CURLOPT_CAPATH, CURLOPT_CERTINFO,
CURLOPT_CHUNK_BGN_FUNCTION, CURLOPT_CHUNK_DATA,
CURLOPT_CHUNK_END_FUNCTION, CURLOPT_CLOSESOCKETDATA,
CURLOPT_CLOSESOCKETFUNCTION, CURLOPT_CONNECTTIMEOUT,
CURLOPT_CONNECTTIMEOUT_MS, CURLOPT_CONNECT_ONLY,
CURLOPT_CONV_FROM_NETWORK_FUNCTION,

CURLOPT_CONV_FROM_UTF8_FUNCTION,
CURLOPT_CONV_TO_NETWORK_FUNCTION, CURLOPT_COOKIE,
CURLOPT_COOKIEFILE, CURLOPT_COOKIEJAR, CURLOPT_COOKIELIST,
CURLOPT_COOKIESESSION, CURLOPT_COPYPOSTFIELDS, CURLOPT_CRLF,
CURLOPT_CRLFILE, CURLOPT_CUSTOMREQUEST, CURLOPT_DEBUGDATA,
CURLOPT_DEBUGFUNCTION, CURLOPT_DIRLISTONLY,
CURLOPT_DNS_CACHE_TIMEOUT, CURLOPT_DNS_SERVERS,
CURLOPT_DNS_USE_GLOBAL_CACHE, CURLOPT_EGDSOCKET,
CURLOPT_ERRORBUFFER, CURLOPT_FAILONERROR, CURLOPT_FILETIME,
CURLOPT_FNMATCH_DATA, CURLOPT_FNMATCH_FUNCTION,
CURLOPT_FOLLOWLOCATION, CURLOPT_FORBID_REUSE,
CURLOPT_FRESH_CONNECT, CURLOPT_FTPPORT, CURLOPT_FTPSSLAUTH,
CURLOPT_FTP_ACCOUNT, CURLOPT_FTP_ALTERNATIVE_TO_USER,
CURLOPT_FTP_CREATE_MISSING_DIRS, CURLOPT_FTP_FILEMETHOD,
CURLOPT_FTP_RESPONSE_TIMEOUT, CURLOPT_FTP_SKIP_PASV_IP,
CURLOPT_FTP_SSL_CCC, CURLOPT_FTP_USE_EPRT,
CURLOPT_FTP_USE_EPSV, CURLOPT_FTP_USE_PRET,
CURLOPT_GSSAPI_DELEGATION, CURLOPT_HEADER,
CURLOPT_HEADERDATA, CURLOPT_HEADERFUNCTION,
CURLOPT_HTTP200ALIASES, CURLOPT_HTTPAUTH, CURLOPT_HTTPGET,
CURLOPT_HTTPHEADER, CURLOPT_HTTPPOST,
CURLOPT_HTTPPROXYTUNNEL, CURLOPT_HTTP_CONTENT_DECODING,
CURLOPT_HTTP_TRANSFER_DECODING, CURLOPT_HTTP_VERSION,
CURLOPT_IGNORE_CONTENT_LENGTH, CURLOPT_INFILESIZE,
CURLOPT_INFILESIZE_LARGE, CURLOPT_INTERLEAVEDATA,
CURLOPT_INTERLEAVEFUNCTION, CURLOPT_IOCTLDATA,
CURLOPT_IOCTLFUNCTION, CURLOPT_IPRESOLVE, CURLOPT_ISSUERCERT,
CURLOPT_KEYPASSWD, CURLOPT_KRBLEVEL, CURLOPT_LOCALPORT,
CURLOPT_LOCALPORTRANGE, CURLOPT_LOW_SPEED_LIMIT,
CURLOPT_LOW_SPEED_TIME, CURLOPT_MAIL_FROM,
CURLOPT_MAIL_RCPT, CURLOPT_MAXCONNECTS, CURLOPT_MAXFILESIZE,
CURLOPT_MAXFILESIZE_LARGE, CURLOPT_MAXREDIRS,
CURLOPT_MAX_RECV_SPEED_LARGE,
CURLOPT_MAX_SEND_SPEED_LARGE, CURLOPT_NETRC,
CURLOPT_NETRC_FILE, CURLOPT_NEW_DIRECTORY_PERMS,
CURLOPT_NEW_FILE_PERMS,
CURLOPT_NOBODY, CURLOPT_NOPROGRESS, CURLOPT_NOPROXY,
CURLOPT_NOSIGNAL, CURLOPT_OPENSOCKETDATA,
CURLOPT_OPENSOCKETFUNCTION,
CURLOPT_PASSWORD, CURLOPT_PORT, CURLOPT_POST,

CURLOPT_POSTFIELDS, CURLOPT_POSTFIELDSIZE,
CURLOPT_POSTFIELDSIZE_LARGE,
CURLOPT_POSTQUOTE, CURLOPT_POSTREDIR, CURLOPT_PREQUOTE,
CURLOPT_PRIVATE, CURLOPT_PROGRESSDATA,
CURLOPT_PROGRESSFUNCTION,
CURLOPT_PROTOCOLS, CURLOPT_PROXY, CURLOPT_PROXYAUTH,
CURLOPT_PROXYPASSWORD, CURLOPT_PROXYPORT,
CURLOPT_PROXYTYPE,
CURLOPT_PROXYUSERNAME, CURLOPT_PROXYUSERPWD,
CURLOPT_PROXY_TRANSFER_MODE,
CURLOPT_PUT, CURLOPT_QUOTE, CURLOPT_RANDOM_FILE,
CURLOPT_RANGE, CURLOPT_READDATA, CURLOPT_READFUNCTION,
CURLOPT_REDIR_PROTOCOLS, CURLOPT_REFERER, CURLOPT_RESOLVE,
CURLOPT_RESUME_FROM, CURLOPT_RESUME_FROM_LARGE,
CURLOPT_RTSP_CLIENT_CSEQ,
CURLOPT_RTSP_REQUEST, CURLOPT_RTSP_SERVER_CSEQ,
CURLOPT_RTSP_SESSION_ID,
CURLOPT_RTSP_STREAM_URI, CURLOPT_RTSP_TRANSPORT,
CURLOPT_SEEKDATA,
CURLOPT_SEEKFUNCTION, CURLOPT_SHARE, CURLOPT_SOCKOPTDATA,
CURLOPT_SOCKOPTFUNCTION, CURLOPT_SOCKS5_GSSAPI_NEC,
CURLOPT_SOCKS5_GSSAPI_SERVICE,
CURLOPT_SSH_AUTH_TYPES, CURLOPT_SSH_HOST_PUBLIC_KEY_MD5,
CURLOPT_SSH_KEYDATA,
CURLOPT_SSH_KEYFUNCTION, CURLOPT_SSH_KNOWNHOSTS,
CURLOPT_SSH_PRIVATE_KEYFILE,
CURLOPT_SSH_PUBLIC_KEYFILE, CURLOPT_SSLCERT,
CURLOPT_SSLCERTTYPE,
CURLOPT_SSLENGINE, CURLOPT_SSLENGINE_DEFAULT,
CURLOPT_SSLKEY,
CURLOPT_SSLKEYTYPE, CURLOPT_SSLVERSION,
CURLOPT_SSL_CIPHER_LIST,
CURLOPT_SSL_CTX_DATA, CURLOPT_SSL_CTX_FUNCTION,
CURLOPT_SSL_SESSIONID_CACHE,
CURLOPT_SSL_VERIFYHOST, CURLOPT_SSL_VERIFYPEER,
CURLOPT_STDERR,
CURLOPT_TELNETOPTIONS, CURLOPT_TFTP_BLKSIZE,
CURLOPT_TIMECONDITION,
CURLOPT_TIMEOUT, CURLOPT_TIMEOUT_MS, CURLOPT_TIMEVALUE,

CURLOPT_TLSAUTH_PASSWORD, CURLOPT_TLSAUTH_TYPE,
CURLOPT_TLSAUTH_USERNAME,
CURLOPT_TRANSFERTEXT, CURLOPT_TRANSFER_ENCODING,
CURLOPT_UNRESTRICTED_AUTH,
CURLOPT_UPLOAD, CURLOPT_URL, CURLOPT_USERAGENT,
CURLOPT_USERNAME, CURLOPT_USERPWD, CURLOPT_USE_SSL,
CURLOPT_VERBOSE, CURLOPT_WILDCARDMATCH, CURLOPT_WRITEDATA,
CURLOPT_WRITEFUNCTION, CURLOPT_UNIX_SOCKET_PATH,
CURLOPT_XFERINFODATA,
CURLOPT_XFERINFOFUNCTION, CURLOPT_XOAUTH2_BEARER,
CURLOPT_SSL_ENABLE_ALPN,
CURLOPT_SSL_ENABLE_NPN, CURLOPT_SSL_FALSESTART,
CURLOPT_SSL_OPTIONS,
CURLOPT_SASL_IR, CURLOPT_SERVICE_NAME, CURLOPT_PROXYHEADER,
CURLOPT_PATH_AS_IS, CURLOPT_PINNEDPUBLICKEY, CURLOPT_PIPEWAIT,
CURLOPT_LOGIN_OPTIONS, CURLOPT_INTERFACE, CURLOPT_HEADEROPT,
CURLOPT_DNS_INTERFACE, CURLOPT_DNS_LOCAL_IP4,
CURLOPT_DNS_LOCAL_IP6,
CURLOPT_EXPECT_100_TIMEOUT_MS, CURLOPT_MAIL_AUTH,
CURLOPT_PROXY_SERVICE_NAME,
CURLOPT_TCP_KEEPALIVE, CURLOPT_TCP_KEEPIDLE,
CURLOPT_TCP_KEEPINTVL,
CURLOPT_TCP_NODELAY, CURLOPT_SSL_VERIFYSTATUS

Please visit: http://curl.haxx.se/ to see a documentation about these options.

54.3 Handling Web pages.

There are two different ways to load an HTML page: either through a callback function or with a filename.

Callback

The first possibility is to associate your *url* object with a callback function, whose signature should be the following:

function url_callback(string content,myobject o);

The function will be associated with the following declaration:

url u(o) with url_callback.

In that case, you should use: *url(string html)* as method in order to have each block of texts loaded from your web page. For each block, your *url_callback* will be called with the block content as value.

Example:

```
use("tamgucurl");

function fonc(string content,self o) {
    println(content);
}

curl c with fonc;
//we set a proxy, which will be used as a way to load your web pages through
c.proxy("http://myproxy.mycompany:5050");
//we load our web page. For each block, func will be called...
c.url("http://www.liberation.fr/");
```

File

The other possibility is to provide the *url* method with a filename, which will be used to store the content of your web page. In that case, do not declare your variable with a callback function.

Example:

```
use("tamgucurl");

curl c;
//we set a proxy, which will be used as a way to load your web pages through
c.proxy("http://myproxy.mycompany:5050");
//we load our web page. For each block, func will be called...
c.url("http://www.liberation.fr/","c:\temp\myfile.html");
```

Example:

```
//This example shows how to query a search site (the URL provided here does
not exist at the time when this manual was written)
string mytxt;
function requester(string s,self e) {
    mytxt+=s;
}

curl querying with requester;
//we set a proxy
querying.proxy("my.proxy.com:8080");
//we set some options, which are necessary to proceed with our command
querying.options("CURLOPT_HEADER", 0);
querying.options("CURLOPT_VERBOSE", 0);
querying.options("CURLOPT_AUTOREFERER",1);
querying.options("CURLOPT_FOLLOWLOCATION",1);
querying.options("CURLOPT_COOKIEFILE","");
querying.options("CURLOPT_COOKIEJAR","");
querying.options("CURLOPT_USERAGENT", "Mozilla/4.0 (compatible;)");

function request(svector words) {
    //we build ou query
    string query="http://my.any.search.engine.com/html/?q=";
    mytxt="";
    string thequery=query+words.join("+");
    querying.url(thequery);
    println(mytxt);
}

request(["test","word"]);
```

55 Python library (pytamgu)

The *pytamgu* library is a dual library. It is both a Python and a Tamgu library. You can either execute Python code from within a Tamgu program, or execute a Tamgu program from within Python code.

55.1 As a Tamgu library

It exposes a new type: *python.*

The base Tamgu types: *boolean, int, long, float, fraction, string, vector containers and map containers* are automatically mapped onto the corresponding Python types.

The *python* type exposes the following methods:

1. **close()***: Close the current Python session.*
2. **execute(string funcname,p1,p2...)***: execute a python function with p1,p2 as parameters*
3. **import(string python)***: import a python file*
4. **run(string code)***: Execute python code*
5. **setpath(string path1,string path2 etc...)***: Add system paths to python*

The *setpath* method is crucial to use the *import* method, which works exactly as the *import* keyword in Python. If you want to import a Python program at a specific location, which has not been referenced through PYTHONPATH, you need to add it with *setpath* first.

Example

First we implement a small Python program, which we call: *testpy.py*

```
val="here"

#The input variables are automatically translated from Tamgu into Python variables
def Iteste(s,v):
  v.append(s)
  v.append(val)
  return v
```

Then we implement our own Tamgu program, which will call this file (which we suppose to be in the same directory as our Tamgu program)

```
//We need to use Pytamgu for our own sake
use("pytamgu");

//we need a variable to handle the Python handling
python p;

//we suppose that our Python program is in the same directory as our Tamgu
program
p.setpath(_paths[1]);
//We then import our program
p.import("testpy");

vector v;
string s="kkk";

//We execute the Python function Itest, which takes as input a string and a vector,
//which will converted into Python objects on the fly.
//The output is automatically re-converted into a Tamgu vector (from the Python
vector)
vector vv=p.execute("Itest",s,v);

println(vv); //output is: [‘kkk’,’here’]

p.close(); //we close the session
```

55.2 As a Python library

You can import the pytamgu library, which then will expose two methods.

1. **load(file,arguments,mapping)**
   a. *file is the filename of the Tamgu file to load*
   b. *arguments is a string, which provides arguments to the Tamgu file separated with a space.*

c. *If mapping is one, then a python method is created for each function in the Tamgu file, with the same name as the Tamgu functions.*

d. *This method returns a handle*

2. **execute(handle, function_name,[arg1,arg2…argn])**

a. handle is the handle of the file, which contains the function we want to execute.

b. function_name is the name of the function in the Tamgu file.

c. [arg1…argn] is the list of arguments that will be provided to the Tamgu program as a vector of strings.

If you use the mapping option, the execute method is optional. The values returned by the Tamgu program are automatically translated into Python object, the same applies to the arguments transmitted to the Tamgu program.

Note: Tamgu always returns Python Unicode strings.

Example:

Tamgu Program

```
vector v=[1..10];

function rappel(string s, int j) {
   j+=10;
   v.push(j);
   v.push(s);
   return(v);
}
```

Python Program

```
import pytamgu
```

```
h0=pytamgu.load("rappel.kif","",1)

# we use the mapping to a Python function
v=rappel("Test",10)
for i in v:
	print i

# This is equivalent to
v = pytan.execute(h0, “rappel”, [“Test”,10])
```

56 Library LINEAR: type linear

Tamgu provides also an encapsulation of the *liblinear* library (see http://www.csie.ntu.edu.tw/~cjlin/liblinear/ for more information).
This library is used to implement classifiers and exposes the following methods.

The name of the library is *linear:* use("linear");

56.1 Methods

1. **cleandata()**: *clean internal data*
2. **crossvalidation()**: *Relaunch the cross validation with new parameters. The result is a fmap.*
3. **loadmodel(string filename)**: *Load your model. A model can also automatically be loaded with a constructor.*
4. **options(smap actions)**: *Set the training options (see below)*
5. **predict(fvector labels, vector data,bool predict_probability,bool infos)**: *Predict from a vector of treemapif. labels is optional. When it is provided, it is used to test the predicted label against the target label stored in labels. If infos is true, the first element of this vector is an info map.*
6. **predictfromfile(string input,bool predict_probability,bool infos)**: *Predict from a file input. The result is a vector. If infos is true, the first element of this vector is an info map.*
7. **savemodel(string outputfilename)**: *save your model in a file*
8. **trainingset(fvector labels,vector data)**: *create your training set out of a treemapif vector.*
9. **train(string inputdata,smap options,string outputfilename)**: *train your training data with some options. outputfilename is optional. It will be used to store the final model if provided (the method can also be called with the name load).*

56.2 Training options

The training options should be provided as a smap, with the following keys: *s,e,c,p,B,wi,M and v.*

1. 's' type : set type of solver (default 1)
    a. for multiclass classification

    0 -- L2-regularized logistic regression (primal)

    1 -- L2-regularized L2-loss support vector classification (dual)

    2 -- L2-regularized L2-loss support vector classification (primal)

    3 -- L2-regularized L1-loss support vector classification (dual)

    4 -- support vector classification by Crammer and Singer

    5 -- L1-regularized L2-loss support vector classification

    6 -- L1-regularized logistic regression

    7 -- L2-regularized logistic regression (dual)

    b. for regression

    11 -- L2-regularized L2-loss support vector regression (primal)

    12 -- L2-regularized L2-loss support vector regression (dual)

    13 -- L2-regularized L1-loss support vector regression (dual)
2. 'c' cost : set the parameter C (default 1)
3. 'p' epsilon : set the epsilon in loss function of SVR (default 0.1)
4. 'e' epsilon : set tolerance of termination criterion
    a. 's' 0 and 2

    |f'(w)|_2 <= eps*min(pos,neg)/l*|f'(w0)|_2,where f is the primal function and pos/neg are # of positive/negative data (default 0.01)

    b. 's' 11

    |f'(w)|_2 <= eps*|f'(w0)|_2 (default 0.001)

    c. 's' 1, 3, 4, and 7

    Dual maximal violation <= eps; similar to libsvm (default 0.1)

    d. 's' 5 and 6

    |f'(w)|_1 <= eps*min(pos,neg)/l*|f'(w0)|_1,where f is the primal function (default 0.01)

    e. 's' 12 and 13

    |f'(alpha)|_1 <= eps |f'(alpha0)|, where f is the dual function (default 0.1)
5. 'B' bias : if bias >= 0, instance x becomes [x; bias]; if < 0, no bias term added (default -1)

6. 'wi' weight: weights adjust the parameter C of different classes. ‘i' stands here for an index. A key might look like “w10” for instance.
7.
8. 'M' type: type of multiclass classification (default 0)
    a. 'M' 0: one-versus-all
    b. 'M' 1: one-versus-one
9. 'v' n: n-fold cross validation mode

Note that it is possible to use the following strings instead of integers for the solver:

- "L2R_LR" is 0
- "L2R_L2LOSS_SVC_DUAL" is 1
- "L2R_L2LOSS_SVC" is 2
- "L2R_L1LOSS_SVC_DUAL" is 3
- "MCSVM_CS" is 4
- "L1R_L2LOSS_SVC" is 5
- "L1R_LR" is 6
- "L2R_LR_DUAL" is 7
- "L2R_L2LOSS_SVR = 11" is 8
- "L2R_L2LOSS_SVR_DUAL" is 9
- "L2R_L1LOSS_SVR_DUAL" is 10

Example:

```
maps s={'s':'L1R_LR','B':1,'v':9};
```

The input structure to both *predict* and *trainingset*

These two methods accept as input two structures. The first one is a *fvector*, which will contain the so-called labels (float elements), the second one is a vector of *treemapif*. The two structures should have exactly the same size.

Each element in the second parameter vector is a *treemapif*, where the key is the index and the value the associated probability. This structure has been chosen to store sparse vectors.

The predict methods output

The two predict methods output is a vector of maps. The first element is an *info* smap, which contains some measures over the whole analysis (such as the “accuracy”). The next elements depending on the flag: *predict_probability* are either the predicted label or a map containing for each line from the input structure the label with the associated list of probabilities.

With predict probability

{'1':[0.999725,3.66243e-05,4.85055e-06,4.49336e-07,6.43783e-05]}

The key is the chosen label, with the list of its probabilities.

Without predict probability

It is simply the chosen label.

56.3 Examples

Training example

```
use("linear");

//we load the library
use("liblinear");

string trainFile = "output.dat";

//we declare a liblinear variable
liblinear train;

//We set the options
map options ={"c":100, "s":'L2R_LR', "B":1, "e":0.01};

//we load our model, whose training output will be stored in the model_test file
train.load(trainFile, options, "model_test" );
```

Predict example

```
use("linear");

//The input file
string testFile = "trainData.dat";

//a liblinear variable, which is declared with its model (we could use loadmodel
instead)
```

```
liblinear predict("model_test");

//The prediction is done from a file
vector result = predict.predictfromfile(testFile,true);
```

57 Library wapiti: type wapiti

libwapiti is an encapsulation of the wapiti library, which is available on:

http://wapiti.limsi.fr .
Copyright (c) 2009-2013 CNRS
All rights reserved.

wapiti provides an efficient implementation of the CRF method, to do tagging or entity extraction. If you need any information on the system please refer to the manual on: http://wapiti.limsi.fr/manual.html

The name of the library is *wapiti:* use('wapiti');

libwapiti exposes the following methods.

57.1 Methods

1. **loadmodel(string crfmodel)**: *Loading a CRF model.*
2. **options(svector options)**: *Setting options. See below for the available options. Options should be place in the svector as used on the command line of wapiti.*
3. **train()**: *Launch training. Requires options to have been set in advance.*
4. **label(vector words)**: *Launch labelling for a vector of tokens. Returns a vector of labels for each token.*
5. **lasterror()**: *Return the last error, that did occur.*

57.2 Options

Wapiti exposes some options to deal with all the possibilities engrained in the system. Below is a list of these options, which should be supplied as a *svector*, exactly as these options would be provided with the command line version of wapiti.

- Train mode:
- train [options] [input data] [model file]

- –me force maxent mode
- –T | --type STRING type of model to train
- –a | --algo STRING training algorithm to use
- –p | --pattern FILE patterns for extracting features
- –m | --model FILE model file to preload
- –d | --devel FILE development datset
- –rstate FILE optimizer state to restore
- –sstate FILE optimizer state to save
- –c | --compact compact model after training
- –t | --nthread INT number of worker threads
- –j | --jobsize INT job size for worker threads
- –s | --sparse enable sparse forward/backward
- –I | --maxiter INT maximum number of iterations
- -1 | --rho1 FLOAT l1 penalty parameter
- -2 | --rho2 FLOAT l2 penalty parameter
- –o | --objwin INT convergence window size
- –w | --stopwin INT stop window size
- –e | --stopeps FLOAT stop epsilon value
- –clip (l-bfgs) clip gradient
- –histsz INT (l-bfgs) history size
- –maxls INT (l-bfgs) max linesearch iters
- --eta0 FLOAT (sgd-l1) learning rate
- –alpha FLOAT (sgd-l1) exp decay parameter
- –kappa FLOAT (bcd) stability parameter
- –stpmin FLOAT (rprop) minimum step size
- –stpmax FLOAT (rprop) maximum step size
- –stpinc FLOAT (rprop) step increment factor
- –stpdec FLOAT (rprop) step decrement factor
- –cutoff (rprop) alternate projection

- Label mode:
- label [options] [input data] [output data]
  - --me force maxent mode
  - –m | --model FILE model file to load
  - –l | --label output only labels
  - –c | --check input is already labeled

- –s | --score add scores to output
- –p | --post label using posteriors
- –n | --nbest INT output n-best list
- --force use forced decoding

57.3 Training

To train a CRF, you need a text file with annotations, where each line is a token with its tags separated with a tab.

Example:

```
UNITED STATES	NOUN	LOCATION_b
SECURITIES	NOUN	ORGANISATION_i
AND	CONJ	ORGANISATION_i
EXCHANGE	NOUN	ORGANISATION_i
COMMISSION	NOUN	ORGANISATION_i
Washington	NOUN	ORGANISATION_i
,	PUNCT	ORGANISATION_i
D.C.	NOUN	LOCATION_b
20549	DIG	NUMBER_b
FORM	VERB	null
N	NOUN	null
```

In this example, we have a token for each line associated with two different tags.

N.B. The tag “null” in this example is a simple string that does not have a specific interpretation except for this specific example.

Pattern file

You also a need a “pattern” file, which would be implemented according to the manual either of CRF++ (on which it is based, see *http://taku910.github.io/crfpp/* for more information) or as described in *http://wapiti.limsi.fr/manual.html*.

Example:

```
# Unigram
U00:%x[-2,0]
U01:%x[-1,0]
U02:%x[0,0]
U03:%x[1,0]
U04:%x[2,0]
```

```
U05:%x[-2,1]
U06:%x[-1,1]
U07:%x[0,1]
U08:%x[1,1]
U09:%x[2,1]
U10:%x[-2,0]/%x[0,0]
U11:%x[-1,0]/%x[0,0]
U12:%x[0,0]/%x[1,0]
U13:%x[0,0]/%x[2,0]
U14:%x[-2,1]/%x[0,1]
U15:%x[-1,1]/%x[0,1]
U16:%x[0,1]/%x[1,1]
U17:%x[0,1]/%x[2,1]

# Bigram
B
```

Program

Here is a small program, which takes as input a training file and a pattern file to produce the model that will be used to label entities.

```
use('wapiti');

wapiti tst;

//we are going to produce our model, based on the pattern file and the training file
svector v=["train","-p","pattern","-1","5","training","model"];

tst.options(v);

tst.train();
```

57.4 Labeling

The labeling is processed through the method: "*label*". In order to use this method, you must first load the model that you produce through training. The process consists in sending a list of tokens and receiving as output a vector, with the same size, containing the corresponding labels. Actually, you need to provide the system with a list of tokens, each associated to their specific tag (here a POS). The system will then try to evaluate the final tag for each of them according to the training set.

Example

```
use('wapiti');


//Our input...
svector words=['Growth  NOUN','&       CONJ','Income  NOUN',
    'Fund    NOUN','(       PUNCT','Exact   ADJ',
    'name    NOUN','of      PREP','registrant      NOUN',
    'as      PREP','specified       ADJ','in      PREP','charter NOUN'];

wapiti tst;
//We then load our model...
tst.loadmodel("model");
//We then label our vector of tokens
svector res=tst.label(words);
//Which returns as output a list of tags...
println(res);
```

The result is : ['ORGANISATION_b', 'ORGANISATION_i', 'ORGANISATION_i', 'ORGANISATION_i', 'null', 'null', 'null', 'null', 'null', 'null', 'null', 'null', 'null']

58 Library word2vec: type word2vec

Tamgu provides an encapsulation of word2vec. See https://code.google.com/p/word2vec/ for more information.

With this library you can both train the system on corpora and use the result through *distance* or *analogy.*

The name of the library is *word2vec*: use("word2vec");

58.1 Methods

1. **accuracy(vector words,int threshold)**: *Finding accuracies for a vector of many times 4 words. Return a fmap. If threshold is not supplied then its value is 30000*
2. **analogy(svector words)**: *Finding analogies for a group of words. Return a fmap*
3. **distance(svector words)**: *Finding the distance in a vector of words. Return a fmap.*
4. **features()**: *Return a map of the vocabulary with their feature values.*
5. **initialization(map m)**: *Initialization of a word2vec training set*
6. **loadmodel(string filename,bool normalize)**: *Loading a model*
7. **trainmodel(vector v)**: *Launching the training. If v is not supplied, then the system utilizes the input file given in the initialisation options*
8. **vocabulary()**: *Return a itreemap of the vocabulary covered by the training.*

58.2 Options

The options are supplied as a map to the library. These options are exactly the same as the one expected by the library.

```
map options={"train":input.txt,"output":output.txt,"cbow":1,
   "size": 200,"window":5,"negative":25,"hs":0,
"sample":1e-4,"threads":20,"binary":1,"iter":15};
```

For a better explanation of these options, please read the appropriate information on Word2Vec site. The most important options are:

- “train”: this option should be associated with the file that will be used as training material.
- “output”: the value for that key is the output file, in which the final training model will be stored.
- “window”: this value defines the number of words taken into account as a proper context for a given token.
- “threads”: word2vec utilizes threads to speed up the process. You can define the number of threads the system can use.
- “size”: this value defines the size of the vector that is associated to each token.
- “iter”: this value defines the number of iterations to build the model.

Once, these options have been supplied, call *initialisation* to set them in.

Example:

```
//We will train our system on input.txt, the result will be stored in output.bin
use("word2vec");

word2vec wrd;

//Window will be 5 words around the main word.
//Vector size for each word will be 200
//The system will use 20 threads to compute the final model
//with 15 iterations.

map options={"train":input.txt,"output":output.bin,"cbow":1,
   "size": 200,"window":5,"negative":25,"hs":0,
"sample":1e-4,"threads":20,"binary":1,"iter":15};

wrd.initialization(options);
wrd.trainmodel();
```

58.3 Usage

To use a model, once it has been created, you simply use *loadmodel*, then you can use either distance, analogy or accuracy. All these methods returns a list of words with their distance to the words in the input vectors. The vocabulary against which the words are compared is the one extracted from the input document. You can have access to all these words with the function *vocabulary.*

Example:

```
use("word2vec");

word2vec wrd;

//we load the model that was obtained through training
wrd.loadmodel("output.bin");
svector v=[“word”];

fmap res=wrd.distance(v);
```

58.4 Type w2vector

Each word extracted from the input document is associated with a specific vector whose size is supplied at training time with the option: “*size”.* In our example, this size is set to 200.

It is actually possible to extract a specific vector from the training vocabulary and store it into a specific object: *w2vector.*

Methods

1. **dot(element)**: *Return the dot product between two words. Element is either a string or a w2vector.*
2. **cosine(element)**: *Return the cosine distance between two words. Element is either a string or a w2vector.*

3. **distance(element)**: *Return the distance between two words. Element is either a string or a w2vector.*

4. **threshold(element)**: *Return or set the threshold.*

5. **norm(element)**: *Return the vector norm.*

Creation

The initialization of a w2vector object is done requires first that a model has been loaded, then you need to provide both the token string and a threshold.

Example:

```
use("word2vec");

w2vector wrd;

//we load the model that was obtained through training
wrd.loadmodel("output.bin");

w2vector w=wrd["tsunami":0.5];
w2vector ww=wrd["earthquake":0.5];

println(w.distance(ww));
```

The threshold is not mandatory. It is actually used when you compare two w2vector elements together to see whether they are close. The threshold is then used to detect whether the distance between the two elements is superior to that threshold.
In other words:

if (w==ww)… is equivalent to if (w.distance(ww)>=w.threshold())

Example:

```
if (w==ww)
   println("ok");

if (w=="earthquake")  //we can compare against a simple string…
   println("ok");
```

fvector

It is also possible to retrieve the inner float vector from a w2vector…

```
fvector vvv=w;
```

vvv is:

```
[0.049775, -0.0498451, -0.0722533,0.0536649, -0.000515156,-0.0947062,
0.0294775,-0.0146792,-0.100351,0.0480318,0.071128,0.0268629...]
```