Xerox Incremental Parser

KiF Language

6, CH DE MAUPERTUIS
38240 MEYLAN
FRANCE

Printed in France

Author: Claude Roux

E-mail : claude.roux@xrce.xerox.com

6, CH DE MAUPERTUIS
38240 MEYLAN
FRANCE

Contents

Summary

This document describes the KiF language.

Knowledge in Frame: KiF

The KiF language is part of the XIP engine. This interpreted language shares the same variables as the XIP script language. The functions and frames defined in a KiF section can be fully called and integrated in any XIP rules. Furthermore, a KiF program can also be executed from the command line with the XIP engine, without any grammars.
The KiF 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.
The most important element of this language is that the interpretation of a variable or of an operator depends on its context.

Some elements

A KiF 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.

IMPORTANT

- KiF is case-sensitive, which is not the case for XIP
- The KiF language is fully compliant with XIP variable declarations.
- A variable declared in XIP is visible as such in KiF.
- Any modification of a XIP variable in a KiF program shows up in XIP script language.

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 predeclarations

The pre-declarations of functions and frames is not necessary in KiF, 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.

System Functions

Exit: _exit(id)

This function is used to exit definitely from a program. If it is used with a GUI, then the exit simply stops the execution, in the case of the command line version, it kills the process and returns the id.

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. You can modify this value to increase the number of threads that can run in parallel.

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.

Garbage information: map m=_gcsize();

KiF provides a function which returns some information about the garbage collector as a map. This map has three values:

- **deleted:** *this is the size of the deleted vector in which free cells from the garbage are kept.*
- **cursor:** *this is the position of the last element in the* deleted *vector which has been reached before calling the garbage collector.*
- **garbage:** *this is the actual size of the garbage collector*
- **cursor…:** *specific garbages are also managed by KiF for strings, integers, floats, vectors, maps and lists. These structures are actually pre-created when KiF is initialized. KiF never actually deletes these elements from memory, but reuses them whenever their reference is back to 0. This explains why the specific cursors might move across the different garbages back and forth in a random way.*

Garbage function: _garbagefunction(function);

KiF provides a simple mechanism to detect and check when the garbage collector is called. When *garbagefunction* is implemented at the beginning of a KiF program, with as a parameter a specific function name, then when the garbage collector is called, this function will be executed.

Example**:**

```
function displaygc() {
        printlnerr("Garbage stats:",gcsize());
}

_garbagefunction(displaygc);
```

Garbage survey: _garbagesurvey()

Returns a map containing garbage information for all objects in memory (strings, integers etc.)

Frame Garbage: _garbageframe()

Returns a map containing garbage information for all frame objects created so far.

KiF version: _version()

Returns a string with version information about KiF.

Mirror display: _mirrordisplay(bool)

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

Passing arguments to a KiF program

A KiF program is usually called with KiF with a list of arguments. Each of these arguments is then available to the KiF program through three specific systems variables: *_args, _current* and *_paths.*

Example:

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

_args: Argument vector

Kif provides a specific variable: “*_args*”, which actually is 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

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

```
//Displaying all paths loaded in memory
iterator it=_paths;

for (it.begin();it.nend();it.next())
        print("Loaded: ",it.value(),"\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 character 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 “\”.

KiF provides a specific variable: *_sep*, which returns the right separator in accordance with the current system.

Console

KiF 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 KiF with: *-console.*

N.B. A console is also available with xip, the option in this case is: -kifconsole.

Basic Types

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

Predefined types

KiF provides many different types, which for some of them, are directly related to the XIP rule language. We will come back with more details on these types later on. Most of them are pretty standard in most programming languages except for the XIP object.

Basic Objects:

self, string, int, float, long, bit, fraction, boolean, time, call, tree

Complex Objects:

vector, map, matrix, file, iterator

XIP Objects (only available from within the XIP interpreter):

node, dependency, xml, fst, rule, graph.

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 in many cases is rather useful. For instance, a frame variable can be declared as

private in order to prevent a client in a server application to have access to it.

Example

```
private test toto;
```

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

function, polynomial, 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.

polynomial

A polynomial function is a numerical function, whose result is automatically indexed on the input parameters. If a specific set of parameters has already been processed, then the value computed for these parameters is extracted and returned in lieu of computing it again.

Example

```
polynomial compute(int j) {
        float k=cos(j)*sin(j)^10;
        return(k);
}
```

The first call to *compute*(10), will trigger KiF to actually compute the value:

-8.39072 for k.

If *compute(10)* is called once again, then the system will use its internal index associated to 10 to return the same value. The formula will not be computed again.

Note

Polynomial automatically builds a map in memory based on the input parameters. These functions consume some resources and should be used with caution.

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]);
}
```

thread

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

Example:

```
function 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 object 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 *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.

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. KIF provides a specific operator for this task: <<<, which is 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. The only constraint is the life of this variable should at least be the one of 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();
```

Multiple definitions

KiF 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
```

Important

To the difference with C++ for instance, KiF does not consider selection ambiguity as an error. KiF will therefore select the *first* function that matches the argument list of the calling function, which means that the order in which functions are declared is important.

Optional Arguments

KiF 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.*

environment() function

A function might be called in a specific context, whose knowledge can be useful in many cases. This context is available through the *environment()* function, which returns this context as a string or as an integer.

Example:

```
//We define our function
function toto(int i) {
   string s=environment();
   print("Environment:",s,"\n");
   return(i+10);
}

//We use it in an expression
int j=toto(10);
string s=toto(10);
```

Run

```
Environment: int
Environment: string
```

Unlimited number of arguments

It is possible to declare a function with an unlimited number of arguments. In this case, the end of the declaration should be “…”. The arguments are then accessible through the name of the function preceded with an “_” as a vector.

Example

```
//we declare a function with an unlimited number of arguments
function test(int i,...) {
	//the other arguments are store in: _test as a vector
	println("Arguments:",_test);
	return;
}

//We call our function
println("Test:",test(14,18,90));
```

Run

Arguments: [18,90]

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 KiF file. If a KiF program is loaded from within another KiF 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 KiF, 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
```

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");
                }
        }
}
```

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();
```

_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++
```

_final function

The *_final* function is called whenever a frame object is deleted by the garbage collector. 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 garbage collector*, 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--;
}
```

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.

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 KiF will yield an error about missing parameters.

Common variables

KiF 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
```

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
```

Sub-framing or enriching a frame

KiF 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);
        }
}
```

Function pre-declaration

Functions can also be pre-declared, and their body can then be defined later.

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

        function display(); //we prepare a display function implementation

}

//We add some code here after…
…

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

frame myframe  {
        function display() { //We actually implement it…
                println(i);
        }
}
```

This is especially useful, if you want two frames to use methods from each other.

Note: It is very important that the order in which functions are pre-declared reflect their actual declaration later on. In other words, if you declare two functions with the same name, then the order in which these two functions will be declared should be the same as for their pre-declaration.

Example

```
frame myframe {

   //we pre-declare two versions of recall
   function recall(int k);
   function recall(string v);

   function Icall(int i) {
      //We call the first one with an integer
      int v=recall(i);
      println(v);
      //the other one with a string
      string u=recall("Recall:");
      println(u);
   }
}

frame myframe {
   //The bodies are declared here...
   //THEY ARE declared in the same order as their pre-declaration.
   //First the integer
   function recall(int i) {
      return(i*4);
   }

   //Then the string
   function recall(string s) {
      return(s+"test");
   }

}
```

Sub-frames

```
…

//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, KiF 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"
```

Cast

The developer can enrich the frame with specific functions that will be used to *cast* a frame into another value. The most common one is *string()* which returns the string interpretation of given frame. However KiF provides every single method to cast into a *float*, an *int*, a *Boolean*, a *map* or a *vector,* but also into another *frame.*

Implementing a *cast* function is quite simple. It is a simple function whose name is the cast itself.

Example:

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

          function string() { //a string cast
```

```
            return(s);
        }

        function int() { //an integer cast
          return(i);
    }
}
```

In the case of a cast between two frames, the name of the function should be the name of that frame.

Example

```
frame frameone  {
        int i=10;        //every new frame will instantiate i with 10
}

frame frametwo  {
    int j=100;        //every new frame will instantiate i with 10

    function frameone() { //we define our cast from frametwo into frameone
          frameone f;
        f.i=j;
           return(f);
    }
}
```

This cast will be used in an instantiation or in a function call to transform on the fly a given frame into another one.

Comparison functions

KiF 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.k)
                    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

KiF 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.
```

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]
```

Associate Functions: *WITH* operator

Frames can be declared with *associate* functions that are called each time a value in a frame is modified. In the case of a frame, three cases can occur:

1. The associate function is defined at the frame level
2. The associate function is defined with a frame variable
3. The associate function is defined at the field level

If an associate function is declared at the frame level, then it is automatically superseded by any functions declared at the variable level, which in turn are superseded by any functions declared at the field level.

Example

```
//Next is a test on frames and associate functions.
frame testcallwith;

function calllocal(testcallwith tx,int before,int after) {
        println("LOCAL",tx,before,after);
}

function callframe(testcallwith tx,int before,int after) {
        println("FRAME",tx,before,after);
}


//We declare a frame with an associate function callframe
frame testcallwith with callframe {
        //a local associate function
        int i with calllocal=10;
        int j=30;

        function string() {
                return(i);
        }
}

function callvariable(testcallwith tx,int before,int after) {
        println("VARIABLE",tx,before,after);
}

//this variable is associated with an associate variable function
testcallwith callt with callvariable;
testcallwith callt2;

callt.i=15; //This modification will trigger calllocal associated with i
callt2.i=20; //This modification will trigger calllocal associated with i
callt.j=15; //This modification will trigger callvariable associated with callt
callt2.j=20; //This modification will trigger callframe associated with testcallwith frame declaration
```

Execution:

```
LOCAL 10 10 15          //modification of callt.i
LOCAL 10 10 20          //modification of callt2.i

VARIABLE 15 30 15       //modification of callt.j
FRAME 20 30 20                  //modification of callt.j
```

Main frame: _KIFMAIN

The variable _KIFMAIN is a specific case of a frame variable: it is the main frame variable of KiF, the one that holds the dictionary in which every single global variable and global function is stored.

_KIFMAIN can be used to create global variable on the fly or to remove a specific instance of an object in memory.

_KIFMAIN indexing

_KIFMAIN works a limited *map*, which means that new items, which are added to the main memory, are simply added through a simple key string: *their actual name.*

Example:

```
//We create a new instance:
int i;
_KIFMAIN["TOTO"]=i;
```

_KIFMAIN.pop("name")

_KIFMAIN also exposes the method: "pop", which can be used to remove an instance of a given variable from memory.

Example:

```
//We delete it:
_KIFMAIN.pop("TOTO");
```

Behaviour as:

- A map, _KIFMAIN returns a {key:value…}, where *key* is an item name and *value* its type.
- A string, it still returns the above map.

Object Broker

_KIFMAIN can be used as an object broker in a server to create new objects on the fly with a given name.

```
//We create a test frame
frame test {
        int i;

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

        function Value() {
                return(i);
        }
```

```
}

//This function creates a new global frame element of type: test
function create(string n,int i) {
        test t(i);
        _KIFMAIN[n]=t;
}

//We create a new instance:
create("TOTO",10);
//It will be possible to refer to that variable through:
println(_KIFMAIN["TOTO"]->Value());//Display: 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.***

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);
```

KiF *Contextual*

KiF is a contextual programming language.

The way a variable is *handled* depends on its *context* of utilisation. 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.

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
```

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
```

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.

Explicit conversion

In the case of a frame, the conversion must be explicitly provided by the user. It consists in adding a specific function whose name matches one of the following types: *string, int, float, long, boolean, vector or map.*

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

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

        function int() {
                return(i);
        }
        function string() {
                return(s);
        }
}


myframe test(10);

//print automatically converts each parameter into a string
print("MYFRAME:",test,"\n");

Run
MYFRAME: initial
```

Predefined Types

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

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

e) **string():** *return the string interpretation of a variable*

f) **int():** *return the integer interpretation of a variable*

g) **float():** *return the float interpretation of a variable*

h) **vector():** *return the vector interpretation of a variable*

i) **map():** *return the map interpretation of a variable*

Transparent Object: *self (*ou *auto)*

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 *auto* can also be used instead of *self*. The introduction of *auto* is a reference to the *auto* type in C++11, which is pretty similar to the way *self* is used in KIF.

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);
```

XIP Regular Expression Formalism

The XIP regular expression formalism has been borrowed from XIP and can be used freely within KiF. It can be used independently from XIP.

The meta-characters

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

%d	stands for any digit
%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

Example:

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

The 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

([x,…,x]*,+): defines a character that can have more than one property

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[%d,bef]	matches a1, ab, ae, af, a0, a9 etc.
7) a[be]+	matches ab, ae, abb, abe, abbbe, aeeeb etc.

Type 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++.

Methods

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

1. **base(int b):** *return the numerical value corresponding to the string in 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 set of characters 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. **dos():** *convert a string in DOS encoding*
7. **dostoutf8():** *convert a DOS string into UTF8 encoding*
8. **evaluate():** *evaluate the meta-characters within a string and return the evaluated string (see below).*
9. **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.*
10. **fill(int nb,string char):** *create a string of* nb chars.
11. **find(string sub,int pos):** *Return the position of substring sub starting at position pos*

12. **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.*

13. **get():** *Read a string from keyboard.*

14. **geterr():** *Catch the current error output. Printerr and printlnerr will be stored in this string variable.*

15. **getstd():** *Catch the current standard output. Print and println will be stored in this string variable.*

16. **html():** *Return the string into an HTML compatible string or as a vector of strings*

17. **insert(i,s):** *insert the string s at i. If i is -1, then insert s between each character in the input string.*

18. **isalpha():***Test if a string only contains only alphabetical characters*

19. **isconsonant():** *Test if a string only contains consonants*

20. **isdigit():** *Test if a string only contains digits*

21. **islower():** *Test if a string only contains lowercase characters*

22. **ispunctuation():** *Test if the string is composed of punctuation signs.*

23. **isupper():***Test if a string only contains uppercase characters*

24. **isutf8():** *Test if a string contains utf8 characters*

25. **isvowel():** *Test if a string only contains only vowels*

26. **last():** *return last character*

27. **latin():** *convert an UTF8 string in LATIN*

28. **left(int nb):** *return the first nb characters of a string*

29. **levenshtein(string s):** *Return the edit distance with s according to Levenshtein algorithm.*

30. **lisp():** *Convert a parenthetic expression into a vector (see below)*

31. **lisp(string opening,string closing):** *Convert a recursive expression using* opening *and* closing *characters as separators (see below)*

32. **lower():** *Return the string in lower characters*

33. **mid(int pos,int nb):** *return the nb characters starting at position pos of a string*

34. **ngrams(int n**): *return a vector of all ngrams of rank n.*

35. **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.*

36. **parse():** *Parse a string as a piece of code and returns the evaluation in a vector.*

37. **pop():** *remove last character*

38. **pop(i):** *remove character at position i*

39. **regex(rgx):** *Return all substrings matching rgx*

40. **regex(rgx):** *Return the position of the substring matching rgx in the string*

41. **regex(rgx):** *Return the substring matching rgx in the string*

42. **regex(rgx**): *Test if the regular expression rgx applies to string*

43. **regexip(rgx):** *Return all substrings matching rgx*

44. **regexip(rgx):** *Return the position of the substring matching rgx in the string*

45. **regexip(rgx):** *Return the substring matching rgx in the string*

46. **regexip(rgx):** *Test if the regular expression rgx applies to string*

47. **removefirst(int nb):** *remove the first nb characters of a string*

48. **removelast(int nb):** *remove the last nb characters of a string*

49. **replace(sub,str):** *Replace the substrings matching sub with str*

50. **replaceregexip(rgx,str):** *Replace the substrings matching rgx with* **str**

51. **replacergx(rgx,str):** *Replace the substrings matching rgx with str*

52. **reverse():** *reverse the string*

53. **rfind(string sub,int pos**): *Return the position of substring sub backward starting at position pos*

54. **right(int nb):** *return the last nb characters of a string*

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

56. **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...*

57. **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”].

58. **splitrgx(rgx**): *Split string with regular expression rgx. Return a svector of substrings. Need regular expression operator (…) to keep substrings.*

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

60. **tags(string o,string c):** *Parse a string as a parenthetic expression, where the opening and closing strings are provided*

61. **tokenize(bool comma,bool separator,bool keepwithdigit):** *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”.* keepwithdigit *enables numbers to be concatenated with the strings next to them as in “3G”.* tokenize *returns a svector. Each of these parameters is optional. When one of these parameters is omitted, then its default value is* false*.*

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

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

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

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

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

67. **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.

Latin Table

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

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

Part 1	*Latin-1 Western European*	Perhaps the most widely used part of ISO/IEC 8859, covering most Western European languages: Danish (partial),[1] Dutch (partial),[2]English , Faeroese, Finnish (partial),[3] French (partial),[3] German, Icelandic, Irish, Italian, Norwegian, Portuguese, Rhaeto-Romanic,Scottish Gaelic, Spanish, Catalan, and Swedish. Languages from other parts of the world are also covered, including: Eastern EuropeanAlbanian, Southeast Asian Indonesian, as well as the African languages Afrikaans and Swahili. The missing euro sign and capital *Ÿ* are in the revised version ISO/IEC 8859-15 (see below). The corresponding IANA character set is ISO-8859-1.
Part 2	*Latin-2 Central European*	Supports those Central and Eastern European languages that use the Latin alphabet, including Bosnian, Polish, Croatian, Czech, Slovak,Slovene, Serbian, and Hungarian. The missing euro sign can be found in version ISO/IEC 8859-16.
Part 3	*Latin-3 South European*	Turkish, Maltese, and Esperanto. Largely superseded by ISO/IEC 8859-9 for Turkish and Unicode for Esperanto.
Part 4	*Latin-4 North European*	Estonian, Latvian, Lithuanian, Greenlandic, and Sami.
Part 5	*Latin/Cyrillic*	Covers mostly Slavic languages that use a Cyrillic alphabet, including Belarusian, Bulgarian, Macedonian, Russian, Serbian, andUkrainian (partial).[4]
Part 6	*Latin/Arabic*	Covers the most common Arabic language characters. Doesn't support other

		languages using the Arabic script. Needs to be BiDi andcursive joining processed for display.
Part 7	*Latin/Greek*	Covers the modern Greek language (monotonic orthography). Can also be used for Ancient Greek written without accents or in monotonic orthography, but lacks the diacritics for polytonic orthography. These were introduced with Unicode.
Part 8	*Latin/Hebrew*	Covers the modern Hebrew alphabet as used in Israel. In practice two different encodings exist, logical order (needs to be BiDi processed for display) and visual (left-to-right) order (in effect, after bidi processing and line breaking).
Part 9	*Latin-5 Turkish*	Largely the same as ISO/IEC 8859-1, replacing the rarely used Icelandic letters with Turkish ones.
Part 10	*Latin-6 Nordic*	a rearrangement of Latin-4. Considered more useful for Nordic languages. Baltic languages use Latin-4 more.
Part 11	*Latin/Thai*	Contains characters needed for the Thai language. Virtually identical to TIS 620.
Part 12	*Latin/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.

Meta-characters

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

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

KiF 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.

Operators

sub in s: *test if* sub *is a substring of* s

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 i^{th} 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, KiF 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.

lisp() or lisp(string opening, string closing)

KiF 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”)
))))))))))).))
```

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

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

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

Example:

KiF 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.lisp('<','>');
```

tags(string opening, string closing)

tags is similar to the *lisp* 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']]];

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.regexip("%d%d%c"); //value=78a

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

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

//REGULAR REGULAR EXPRESSIONS: Not available on all platforms
string rgx='\w+day';
string str="Yooo Wesdenesday Saturday";
vector vrgx=str.regex(rgx);             //['Wesdenesday','Saturday']
string s=str.regex(rgx);    //Wesdenesday
int i=str.regex(rgx);           // position is 5

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

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


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

str="ab(kif12,kif14,kif15,kif16)";
vector v=str.extract(0,"kif",",",")"); //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
```

Type: byte, short, int, float, long

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

Note about the C++ implementation:

int and *float* 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.*

- **Methods:**

1. **#():** *return the bit 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. **bytes():** *return the underlying byte implementation of a value.*
13. **cbrt():** *cubic root.*
14. **chr():** *return the ascii character corresponding to this number as a code.*
15. **cos():** *cosine***.**

16. **cosh():** *hyperbolic cosine.*
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. **get():** *Read a number from keyboard***.**
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.*

Complete list of mathematical functions

KiF 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 KiF as a valid number as long as it is a valid hexadecimal string. A hexadecimal declaration can mix upper or lower characters from the hexadecimal digits: A,B,C,D,E,F.

Operators

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

Example

```
float f;
int i=10;
int j=0xAb45;     //Hexadecimal number

f=i.log(i);               //value= 1
f+=10;            //value=11
f=i%5;            //value=0
```

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*

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.
```

As a Vector

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

Function

You can also associate a function 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**

Type bit

The *bit* type implements a vector of bits, which can be used as way to store numerous Boolean values. A *bit* vector can be transformed into a vector or a map of integers. It can also be iterated. It can also take as a value, an integer, a float or a long.

Note:

When a bit vector is built out of an integer, a float or a long, the bit representation depends on the platform implementation of these values, which might be different from one machine to another.

Methods

1. **#(value)**: *returns the complement of the bit vector*
2. **bit vb(nbbits):** *creates a bit vector of size* nbbits. Nnbits *defines the size in bits of the bit vector. However, since bits are stored in blocks of sixteen bits, the actual size might be larger than the one chosen. For instance,* bit v(25) *will generate a 32 bits vector.*

Operators

<<,>>,&,|,^: *bitwise operators. The "<<" operator (shift left) might extend the length of your bit vector size.*

As a string

It returns a hexadecimal representation of the bit vector.

As a vector or as a map

It returns a vector or a map of integers

As an integer or a float

It returns the transformation into an integer of the first 32 bits.

Example

```
bit test1(62); //we create two bit vectors
bit test2(64);
test1=234; //we initialize the first one with 234
test2=654531;
int i=test2;  //we transform this bit vector into an integer…
vector v=test1; //we transform our bit vector into a vector of integer.
test1=test1 & test2; //we compute the binary AND between test1 and test2
test1<<=7; //shift left
test1=#(test1); // we compute the bit complement of our bit vector
test1[1]=true; //we modify the first bit of our bit vector
println(test1[0],test1[1],test1[2],test1[3]); //we display some bit values
```

Type bits (sparse representation of bits)

The *bits* type implements a map of bits, which can be used as way to store numerous Boolean values. A *bits* map can be transformed into a map of integers. It can be iterated. It can also take as a value, an integer, a float or a long.
The type *bits* implements a sparse representation of bits, to the difference of the type *bit* which implements a vector of all bits. Furthermore, bits are stored on a *64 bits long* in *bits*, while it they are stored over a short in *bit.*

Example:

If we declare the two following variables:

```
bit vbit;
bits mbit;

vbit[120]=1; //Then internally we create 120/16=8 shorts to handle all the necessary bits. The last short has the 8th bit set to 1.

mbit[120]=1 //Then internally we create only one element, whose key is 1 (=120/64), and whose bit 56 is set to 1.
```

Methods

1. **#(value)**: *returns the complement of the bit vector*

Operators

&,|,^: *bitwise operators.*

As a string

It returns a hexadecimal representation of the bit vector.

As a map

It returns a map of integers.

As an integer or a float

It returns the transformation into an integer of the first 32 bits.

Example

```
bits test1(62); //we create two bit vectors
bits test2(64);
test1=234; //we initialize the first one with 234
test2=654531;
int i=test2;  //we transform this bit vector into an integer…
```

```
map m=test1; //we transform our bit vector into a vector of integer.
test1=test1 & test2; //we compute the binary AND between test1 and test2
test1=#(test1); // we compute the bit complement of our bit vector
test1[1]=true; //we modify the first bit of our bit vector
println(test1[0],test1[1],test1[2],test1[3]); //we display some bit values
```

Type fraction

KiF 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.

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 providing 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

KiF 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, KiF simplifies the fraction in order to keep it as small as possible.

As a string, KiF 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
```

Type vector

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

Methods

1. **addtowaiton(t1,t2,...,tn):** *Addind new threads to the thread waiting list for a table (see* waiton*).*
2. **apply(a,b,c…):** *apply all functions passing a,b,c etc. as parameters.*
3. **apply(function,a,b,c):** *apply* function *to all elements in the vector, with the current element being the first parameter of the function, and* a,b,c etc… the next parameters.
4. **bytes():** *return the string matching the bytes stored in the vector. The vector should only contains integers between 0..255.*
5. **clear():** *clean the vector*
6. **editdistance(vector v)** : *edit distance between two vectors*
7. **insert(i,x):** *insert the element x at position i*
8. **join(string sep):** *concatenate each element in the vector in a string where each element is separated from the others with sep.*
9. **json():** *return a* json *compatible string*
10. **last():** *return the last element of the vector.*
11. **merge(vector v):** *merge a vector v into the current vector.*
12. **move(int pfrom,int pto):** *move an element from pfrom to pto*
13. **permute()**: *permute the values in a vector. Return true, if other permutations are still available.*
14. **pop():** *remove the last element from the vector.*
15. **pop(int i):** *remove the ith element from the vector.*
16. **pop(string s):** *remove all of occurrences of string s in the vector.*
17. **product():** *Multiply each element with the others*

18. **push(a):** *add* a *to the vector*

19. **range(first,last):** *generate a vector of elements from* first *to* last, *with a step of 1. The type of* first *and* last *defines the content of the vector.*

20. **range(first,last,step):** *generate a vector of elements from* first *to* last, *with a step of* step*. The type of* first *and* last *defines the content of the vector.*

21. **reserve(int nb)**: *reserve nb characters in the vector*

22. **reverse():** *reverse the order of the elements in the vector*

23. **size():** *return the length of the vector*

24. **shuffle():** *reshuffle values in the vector*

25. **sort(compare):** *sort the content of the vector according to* **compare** *function.*

26. **sortfloat(bool order):** *sort the content of the vector, forcing each value to be a float. order==false order is increasing, order=true order is decreasing.*

27. **sortint(bool order):** *sort the content of the vector, forcing each value to be a int. order==false order is increasing, order=true order is decreasing.*

28. **sortlong(bool order):** *sort the content of the vector, forcing each value to be a long. order==false order is increasing, order=true order is decreasing.*

29. **sortstring(bool order):** *sort the content of the vector, forcing each value to be a string. order==false order is increasing, order=true order is decreasing.*

30. **sum():** *Sum each element with the others*

31. **test(int i):** *test if i is a valid slot in the vector*

32. **totext():** *return the string matching the bytes stored in the vector. The vector should only contains integers between 0..255.*

33. **trigger(bool reinit)**: *(only for type* table*) Trigger the execution of the list of threads stored with* waiton*. If* reinit *is true then the threads are all put back in wait state, otherwise they are terminated.*

34. **unique():** *return a vector where duplicates have been removed*

35. **waiton(t1,..,tn,p1,..,p2):** *(only for type* table*) Waiting for threads, with the current table as the first argument for of each thread. p1,p2..pn are optional variables that will be passed as next parameters to each thread. (See below for an example)*

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']
```

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*

:: : *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 i^{th} character of a vector*

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

Extracting variables from a vector

KiF provides a very peculiar method to benefit 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

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

vector myvect;
iterator it;
myvect=[10,5,20];
myvect.sort(compare);
it=myvect;
for (it.begin();it.nend();it.next())
   print("Content:"+it.key()+"="+it.value(),"\n");

RUN
Content:0=5
Content:1=10
Content:2=20
```

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

//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(string i,string j) {
        if (i<j)
            return(true);
        return(false);
}

vector myvect;
iterator it;
myvect=[10,5,20];
myvect.sort(compare);
it=myvect;
for (it.begin();it.nend();it.next())
        print("Content:"+it.key()+"="+it.value(),"\n");

RUN (This time we sort out strings)
Content:0=10
Content:1=20
Content:2=5
```

Example: modification of each element of a vector with a function

It is possible in KiF to call *apply* with as a first parameter a function or a *call* variable. Then, each element from the vector can be called and modified if necessary accordingly.

```
//We first implement a method, with the first element being
//extracted from the vector
function modify(self vectorElement,string s) {
        println(s, vectorElement);
        //we modify it...: a self element is akin to a pointer parameter passing
        vectorElement+=1;
}

//We create a vector, with numerical values
vector v=[1,2,3,4,5];
//We apply our function to each element
v.apply(modify,"Modification");
//we print it out
println("New:",v);

RUN:
//modify has been called 5 times, hence the 5 "modification" below
Modification 1
Modification 2
Modification 3
Modification 4
Modification 5
New: [2,3,4,5,6] //Each value has been modified
```

A simple factorial example

```
//x is the value from the vector, and d a parameter passed to 'apply'
//There are both declared as self, so as to allow their values to be modified
function fact(self x,self d) {
    x*=d;
    d=x;
}

int d=1;
ivector iv=[1,2,3,4,5,6];
iv.apply(fact,d); //iv=[1,2,6,24,120,720]
```

Example *waiton, trigger,* (only for type *table*)

```
//We declare a list of threads, with some parameters
thread f1(table t,vector res,string s) {
    if (t[0]==1)
        println(1,t);
    else
        res.push(1);
}

thread f2(table t,vector res,string s) {
    if (t[0]==2)
        println(2,t);
    else
        res.push(2);
}
```

```
thread f3(table t,vector res,string s) {
    if (t[0]==3)
        println(3,t);
    else
        res.push(3);
}

//we create a table of size 10
table toto(10);

//Two variables that will be added as arguments to the threads
vector r;
string s;

//We initialize the first element with the value: 2
toto[0]=2;
//Then we push all the above threads in a waiting list, with r,s as supplementary
arguments
toto.waiton(f1,f2,r,s);
//triggering of the threads, they will be available for a new launch
toto.trigger(true);
toto.addtowaiton(f3);
//We change the value, then we trigger the threads and terminate them
toto[0]=3;
toto.trigger(false);
```

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.

Methods

1. **apply(a,b,c…):** *apply all functions stored in the list, passing a,b,c etc. as parameters.*
2. **apply(function,a,b,c):** *apply* function *to all elements in the list, with the current element being the first parameter of the function, and* a,b,c etc… the next parameters.
3. **clear():** *clean the list*
4. **editdistance(list v)** : *edit distance between two lists*
5. **first():** *return the first element of the list*
6. **insert(i,x):** *insert the element x at position i*
7. **join(string sep):** *concatenate each element in the list in a string where each element is separated from the others with sep*
8. **json():** *return a json compatible string*
9. **last():** *return the last element of the list*
10. **merge(list l):** *merge a list into the current list.*
11. **permute()**: *permute the values in a vector. Return true, if other permutations are still available.*
12. **pop():** *remove the last element from the list.*
13. **pop(int i):** *remove the ith element from the list.*
14. **pop(string s):** *remove all of occurrences of string s in the list.*
15. **popfirst():** *remove the first element from the list.*
16. **poplast():** *remove the last element from the list.*
17. **product():** *Multiply each element with the others*

18. **pushfirst(a):** *add* a *to the beginning of the list*
19. **pushlast(a):** *add* a *to the end of the list*
20. **reverse():** *reverse the order of the elements in the list*
21. **shuffle():** *reshuffle values in the list*
22. **size():** *return the length of the list*
23. **sum():** *Sum each element with the others*
24. **test(int i):** *test if i is a valid slot in the list*
25. **unique():** *return a list where duplicates have been removed*

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
```

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

Type bvector, ivector, fvector, svector, uvector

These five types are specialized vector containers for bytes (*bvector),* integers (*ivector*), 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*, KiF 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 KiF 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");
```

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, it is not advised to use *resize* within the context of threads, as the concurrent access to elements might be disrupted, use locks if you actually need to resize it.

waiton, trigger, addtowaiton

This type also accepts the *waiton* and *trigger* methods (see their description in the *vector* description) which allows for a global threads triggering. The reason why these methods are only available to this type is due to the fact *that none of table's methods are protected with an internal lock in multithreading.*

Type tuple

A tuple is a very peculiar type of list, in which elements cannot be modified once the tuple defined. A tuple is defined as a list of elements between parentheses:

```
tuple t=(1,3,4);
```

If you want to define a one element tuple, then the only element should be followed by a comma: (1,)

Most of the other operations on vectors or lists are available to tuple, except the possibility to modify one element from it. Hence, t[0]=10 will raise an error.

Type map (treemap and primemap)

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 KiF 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.
primemap is novel sort of 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.

Methods

1. **apply(a,b,c…):** *apply all functions stored in the map, using a,b,c etc. as parameters.*
2. **apply(function,a,b,c):** *apply* function *to all elements in the map, with the current element being the first parameter of the function, and* a,b,c etc… the next parameters. The basic function should contain two parameters, one for the key, the second for the value.
3. **clear():** *clean the vector*
4. **editdistance(map v)** : *edit distance between two maps*
5. **items():***return a vector of key:value.*
6. **invert():** *return a new map where keys and values are switched. The values are now the keys, and the keys the values.*
7. **join(string keysep,string sep):** *merge the keys and their values into a single string.* Keys *are separated from their values with* keysep*, while* key-values *are separated with* sep*.*
8. **json():** *return a json compatible string*
9. **keys():** *returns the map keys as a vector*
10. **merge(map m):** *merge a map into another.*
11. **pop(string key):** *remove the elements matching key*

12. **seeds(ivector primes):** *only for primemaps. Provides the list of keys that are used to create the various levels of hash tables.*

13. **sort(function f) :** *sort a map along a function f and returns a* primemap *object (see below for an example)*

14. **product():** *Multiply each element with the others*

15. **size():** *return the length of the map*

16. **sum():** *Sum each element with the others*

17. **test(string k):** *test is k is a valid key in the map*

18. **values():** *return the 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 *values* of the map, not the keys. Use *test* to perform a test on the keys.

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}
```

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 ");
```

Sorting a map

If you want to sort out a map, you need to provide a function, which will take two parameters as input. Each parameter is a vector of two elements, a key and its value from the map.

Example:

```
//We initialize a map with some value
map m={"a":7,"b":5,"c":4};
//e and v are two vectors of two values.
//The first value of e (respectively v) is a key from the map
//The second value of e (respectively v) is the value associated to this key
//In this case, we sort out along the values
function sorting(self e,self v) {
   if (e[1]<v[1])
      return(true);
   return(false);
}
//The recipient variable must be a primemap object
primemap mp=m.sort(sorting); //The result is: {'c':4,'b':5,'a':7}
```

Specialized maps

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

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, treemaps and primemaps.*

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.

Note

There are still some older naming convention, which for historical reasons are still recognized by KiF:

mapss	*can also be written: smap*
mapsi	*can also be written: imap*
mapsf	*can also be written: fmap*

(The same applies to *primemap* and *treemap*)

Also the previous pattern: **[s|i|f|u][s|i|f|u](tree|prime)map** is still recognized.

Hence: **mapss** *can also be written ssmap*.

N.B. You cannot mix *string* and *ustring* in the same map. Hence, *mapsu* or *mapus* n’existent pas.

Type set, sset, iset, fset, uset

These types are used to handle sets of elements, which you can either store or retrieve.
Note: They are based on the STL std::set implementation.

The first type is a general one that can be used to store any types of data. The three others are specialized to store strings, integers or floats.

Methods

1. **bytes():** *return the string matching the bytes stored in the set. The set should only contains integers between 0..255.*
2. **clear():** *clean the set*
3. **editdistance(set v)** : *edit distance between two sets*
4. **join(string sep):** *concatenate each element in the set in a string where each element is separated from the others with sep.*
5. **json():** *return a* json *compatible string*
6. **merge(set v):** *merge a set v into the current set.*
7. **pop(value):** *remove* value *from the set.*
8. **product():** *Multiply each element with the others*
9. **push(a):** *add* a *to the set*
10. **size():** *return the length of the set*
11. **sum():** *Sum each element with the others*
12. **test(value):** *test if* value *exists in the set*

Initialization

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

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

Operators

x in vset: *return true or a set of indexes, according to the receiving variable. If the set 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 vset) {…}: *loop among all values. At each iteration* s *contains a* value *from* vset.

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

&,|: *intersection or union of two sets*

As an integer or a Float

It returns the size of the set

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 set.

Indexes

You can use indexes with *set* objects, as with vector. However, indexes in sets only checks whether a value exists.

Example

```
set vset=[1,2,3,4,5];

vset.push(10);
vset.push(20);
vset.pop(10); //remove the value 10 from the set

vector v=vset; //transform a set into a vector
```

Type matrix

This type is used to handle matrices. You define the matrix size at creation time and you can store element 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*

Methods

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

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

float angle=56;

function loading(matrix 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);

matrix 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;

matrix inv;

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

Logical Operators on value containers: &,|,^

The value containers are the specific implementation of vectors and maps for strings, floats and integers. If you use logical operators with these containers, then the way they function 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]
```

Type automaton

The type “automaton” provides a very efficient way to handle word lists. A word list is a text file, in which each line exposes one single word:

Example:

A word file should contain lines such as the example below:

Abbeville
Abbies
Abbie
Abbies
Abbie
Abbies
Abbie
Abbis
Abbi
...

This automaton is very efficient to find out if a word belongs to the lexicon or not. It can store hundred of thousand of words. It can also be used in conjunction with a *function to control the way the automaton can be traversed...*

Methods

It exposes the following methods:

1. **add(string wrd):** *add a word to the automaton*
2. **add(string wrd,string lemmafeature):** *add a word and its lemma form in the automaton as in a transducer.*
3. **dump(string pathname):** *dump the content of the automaton in memory into a text file. Useful, if you have been intensively using* addword.
4. **editdistance(string wrd,int threshold,int flags):** *compute an edit distance on the basis of this automaton with a threshold size. The* flags *is a combination of the following action (see below for a example):*

   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.*

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

a_delete: *the automaton can delete a character*

a_insert: *the automaton can insert a character*

a_switch: *the automaton switches two characters*

a_nocase: *the automaton takes into account the difference in case with the current character string to only add 0.1 to the score.*

a_surface: *the automaton returns the value as a surface form.*

a_full: *the automaton returns the surface form concatenated with the lemma forms and features with the feature separator string. Use a "tab" character if the feature separator string is not available.*

a_split: *the automaton will try to cut the string along word boundaries, if the string is a concatenation of different words.*

a_skip: *The skip action is used to "skip" unknown characters aka characters that do not belong to the current set of labels within the automaton. It is different from the a_delete, since it checks that the current character from with the string DO not belong to the valid labels in the automaton. This flag might be used with the a_split action do deal with no letter characters.*

a_track: *the automaton returns in the final vector, a second map that contains for each word the transformations that occurred.*

5. **load (string pathname,string codefeature):** *Load a text file, in which each line exposes one word. The "codefeature" parameter is optional. It is used to detect in a string where the feature section starts.*

6. **loadlemma(string pathname,string codefeature):** *Load a text file in which lines work in pair. On the even line is the surface form and on the odd line is the lemma+features form. The* codefeature *parameter is optional. It is used with the* a_full *flag.*

7. **loadcompact(string pathname,string codefeature):** *Load a file in which the automaton has been stored in a compact*

way (see store*). The* codefeature *parameter is optional. It is used with the* a_full *flag.*

8. **look(string wrd):** *check if a word belongs to the automaton in conjunction with the model function. This function returns a fmap with the selected words as key and their scores.*

9. **look(string wrd,self o):** *check if a word belongs to the automaton in conjunction with the model function. This function returns a fmap with the selected words as key and their scores. The difference with the above version is that it provides an object that replaces the one provided with the definition of the automaton. This object will be the last argument in the call to the associated function.*

10. **model(function):** *defines the model function.*

11. **setcodefeature(string c):** *defines the code which is used to detect the beginning of the feature structure. If it has been already defined with load, then returns an error.*

12. **settransformationmap(map m):** *Set the transformation map to set the weights when traversing the automaton with editdistance*

13. **spotfinalstate(bool):** *Detect final states in the automaton and add either a "$" to the current label if it is a string, or makes it negative otherwise. (see below for model functions).*

14. **store(string c):** *stores a file into a compact mode, which both smaller and faster to load than a text version of that file.*

Declaration

An automaton can be declared with a function, which is used to control the behavior of the automaton character by character.

Two types of functions are available with the following signature:

a) The simplest one is the following:

```
function simple(int c,ivector labels,float score)
```

where:

a) c is the current character (it is negative, when it is the first character in the string) as unicode

b) labels are the next label Unicode codes. ***When a label is associated with a final state in the automaton (a word boundary so to speak), then the code is negative (<0).***

c) score the current score.

b) A more complex function is:

function cpl(string mystr,string base,int current,svector labels,svector actions,float score,self o)

where:

a) mystr is the current string being analyzed

b) base is the current substring extracted so far

c) current is the current character on which the automaton is positioned

d) labels is a vector of the next arcs presented as strings. ***When a label is associated with a final state in the automaton (a word boundary so to speak), then a '$' is appended to the label.***

e) actions is a vector of all the actions that were applied up to now. It does not contain characters which did not undergo any modification.

f) score is the current score

g) is a user provided object (it can have any types)

Return value: fmap or vector

The two functions must return a *fmap or a vector* object.

a) The *fmap* should have as keys the next labels and as values their modified score. In the case of the first function, the labels should be transformed into string first, with the function *ord*, before being stored in the *fmap.* When you build this *fmap*, you must use as keys the values from the *labels* vector. However, if you want the automaton to add or skip a character from the initial string, you can use the "~" value in the key string. The value "~" alone means that the character should be skipped, while "~" concatenated with a label such as "~o" means that the character should be appended in the recognition process.

```
fmap fm={'a':0.9,'b':0.8}
```

b) You can also return a vector of vectors, where each sub-vector contains two elements, the first is the label, the second one is the new score.

```
vector v=[['a',0.9],['b':0.8]];
```

The main difference between the two structures is that the first one does not provide any specific order in which new arcs should be explored.

Process

The process is rather simple. The “look” function traverses the string character by character. For instance, the string “zone” will be analyzed in the following way. The first character is “z”. The system will then search in the automaton an initial state which matches “z”. If such a state exists, then it will call one the above functions, storing in labels all the “arcs” or the characters that can follow a “z” in a dictionary. The user function will then mark for each of these labels which one should be kept, together with a specific score. The process is then recursively applied to each of these labels.

```
z – i – n – g
    o – n – e
        – a – l
```

We have stored for example, in the above automaton the following three words: *zing, zone* and *zonal*.

We want to find a correction for the word: *zong*.

We start with a “z”, the first letter of our word. In this case, in our automaton, we have only a “z” and the system returns only for labels: labels=[‘z’].

We then go to the next character in our string: ‘o’.

As we can see in our automaton, after a “z” only the characters “i” and “o” are available.

The system will then call our function with the following values for “labels”: labels=[‘o’,’i’].

We analyze these labels and we come up with four potential actions:

a) The “o” is correct. It is both in the automaton and is the current character in our string. We add this label to our fmap: res[“o”]=score. We do not need to modify our score in this case, since it is the same value.

b) The “o” is incorrect. It might be a “i”. However, since the two characters are different, then there is a penalty to choose a

"i" instead of a "o". We add to our fmap result: res["i"]=score+1.

c) The "o" is wrong and should be deleted. We add to our fmap result: res["~"]=score+1, since there is a penalty to remove a character from the initial string.

d) We could also add the 'o' and 'i' to the current string, and continue from there on. We then add to our fmap result: res["~i"]=score+1 and res["~o"]=score+1.

Our fmap result is then: res={"o":0, "i":1, "~":1, "~i":1, "~o":1}.

Our look up function will then use each of these labels as a way to explore the rest of the automaton.

i. For instance, if it selects "i" as the next character, then the score will be 1, and there will be a 0 path after to recognize "ng", (see the automaton.)

ii. If it selects "o", then the penalty will be 1 again as "e" and "g" will be different ("zone" and "zong").

If it selects other paths, then the penalty could be even higher.

Declaration

To use one of these functions, you need to declare them with the automaton itself as a "*with*" function or with the *model* method.

```
automaton au(ch) with cpl;
```

Operator in

The "in" operator is only used to detect if a word belongs to the automaton or not. It doesn't work in conjunction with the model function.

Code feature

Features can be appended to the different strings either in the word file or in an "add" function. These features are not excluded from the word detection but are still added to the solutions.

Example

```
automaton au;


au.setcodefeature("+");
au.add("zonking+Sg+Noun");// +Sg+Noun will be isolated from 'zonking'
```

Edit distance

The edit distance function can be used with different actions that can be combined together. Each of these actions can be pretty costly in terms of computational efficiency.

Example

```
automaton au;

au.load("lexicon");
//we combine character change with character deletion and character insertion
//the system will not try these actions on the first character…
au.editdistance("zoning",2,a_change|a_delete|a_insert);
```

Example

```
//This function is the simplest one you can use.
function simple(int c,ivector labels,float score) {
    fmap res;
    //If the score is too large then we leave...
    if (score>2)
        return(res);

    int i;
    string a;
    bool start=false;
    // when the current character is negative, then it means that it is the first
    // character from the string
    if (c<0) {
        start=true;
        c*=-1;
    }

    //If the first character belongs to the current labels, then we only keep
    //this character in the fmap.
    if (start && c in labels)
        res[c.chr()]=0;
    else {
        //we want to put a score on each next arc...
        //We skip the current character
        res["~"]=score+1;
        for (i in labels) {
            //Here we must transform this code into a string
            a=i.chr();
            //We add it to the recognition string
            res["~"+a]=score+1;
            //if the current character is different from one of the possible labels
            if (i!=c)
                //then we replace it
                res[a]=score+1;
            else
                //we do nothing it is the same character
                res[a]=score;
        }
    }
    //we then return the fmap which will be used by the automaton to process the next
character in the string.
    return(res);
}
```

```
//In a certain way, these functions compute an edit distance but based on an actual
automaton to guide the process
function complex(string mystr,string base,
                        int current,svector labels, svector actions,
                        float score,self o) {
    fmap res;
    //If the score is too large we stop
    if (score>2)
        return(res);
    string c=mystr[current];
    string a;
    //If this is the first character in the string and c is in the labels
    if (current==0 && c in labels)
        res[c]=0;
    else {
        //we want to put a score on each next arc...
        //We skip the current character
        res["~"]=score+1;
        for (a in labels) {
            //in this case, a is a string, a label from the automaton
            //We add it to the recognition string
            res["~"+a]=score+1;
            if (a!=c)
                //replacement of a character
                res[a]=score+1;
            else
                res[a]=score;
        }
    }
    //we must return a fmap as a result
    return(res);
}

//The word to check (it does exist)
string ch="zannking";

//we declare our automaton, here with the first function
automaton au(ch) with simple;
//we load our word list file
au.load('C:\XIP\Test\kifcxx\phonologie\FST\data\englishlexiconmin.txt');
//We test if this word belongs to the automaton
if ("zonking" in au)
    println("Ok");

//we use the "simple" function to find the corresponding words.
//the result is a fmap
println(au.look(ch));
```

Type transducer

This type is quite similar in some aspect to the type *automaton,* however it is much more focused on storing and handling lexicons than *automaton* itself. Furthermore, when KiF is used in combination with XIP, the lexicons generated along this type can be directly used in grammars for tokenization and morphological analysis.
This type exposes the following methods:

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):** *Searching for a surface form, which matches a lemma plus features.*

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

6. **lookup(string wrd, nombre threshold,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 switches two characters*

f. **a_nocase:** *the automaton takes into account the difference in case with the current character string to only add 0.1 to the score.*

7. **process(string sentence):** *Analyse a sequence of words using a transducer*

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

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 will 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.

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*

process

process 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.process(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.

Regular Expressions

The regular expressions processed by transducer are very limited:

%c:	defines a character, c is a UTF8 character …
$.. :	defines a string
c0-cn:	defines an interval between two Unicode characters: 0-9
[..]:	defines a list of characters or character intervals.
{…}:	defines a list of strings
.+:	structure should occur at least once.
(..):	defines an optional structure
!n:	inserts a features structure along its number in the feature vector (n>=1).

Examples:

```
transducer t;

//This expression recognizes Roman Numbers
t.compilergx("[D M C L X V I]+!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([e E]([- +])[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-ω 0-9]+!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));           //αβγδεζηθικλμνξοπρςστυφχψω +Greek
```

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.

Methods

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

9. **load(rule):** *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.*

10. **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.*

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.*

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
%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.*
- 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
%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']}]}]}

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']}]}

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']]

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']]

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

```
//evalute 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);
```

Type tree

This object is used to handle a tree structure. It provides the following methods:

Methods

1. **daughter():** *return the first tree node as a* tree *object*
2. **daughter(tree n):** *Add n as the first daughter to the current node*
3. **depth():** *Return the depth of the node in the tree*
4. **editdistance(tree v)** : *edit distance between two trees*
5. **isolate():** *Extract the current node from its tree*
6. **lastnode():** *return the last child tree node as a* tree *object*
7. **lastnode(tree n):** *Test if the current node is the last child of* n
8. **sister(tree n):** *add n as a sister node to the current node*
9. **sister():** *return the next tree node as a* tree *object*
10. **mother():** *return the parent tree node as a* tree *object*
11. **mother(tree n):** *Test if the current node is a parent of* n
12. **previous():** *return the first tree node as a* tree *object*
13. **previous(tree n):** *add n as the previous node*
14. **prune():** *Delete the current sub-tree from the global tree*
15. **tree n=100:** *modify the value of a tree node with anything, here with an integer, but it could any object.*
16. **tree(value):** *create a tree node, with* value *as a value.* Value *can have any types*

Operator

x in tree: *return true or a list of tree nodes, according to the receiving variable.*

for (s in tree) {…}: *loop among all keys.*

As a string

Return the tree value as a string

As an integer or a float

Return the tree value as a integer or a float

Example

```
//Recursive traversing of a tree
function treedisplay(tree t) {
        if (t==null)
                return;
        print(t," ");
        if (t.daughter()!=null) {
                print("("); //subnodes are displayed between “(…)”
                treedisplay(t. daughter());
                print(")");

        }
        treedisplay(t.sister());
}

//we create five nodes, with numerical values
tree test1(1);
tree test2(2);
tree test3(3);
tree test4(4);
tree test5(5);

test1.daughter(test2);
test1.daughter(test3);
test2.daughter(test4);
test4.sister(test5);

treedisplay(test1); //we display now: 1 (2 (4 5 )3 )

//We modify the value of test5
test5=[100,200];

treedisplay(test1); //we display now: 1 (2 (4 [100,200] )3 )

//we remove test4
test4.prune();
treedisplay(test1); //we display now: 1 (2 ([100,200] )3 )


//we use our values to add
int cpt=test1+test2+test3;
println(cpt); //we display 6

self u; //As we do not know anything about the values, we use a self
for (u in test1)
  print(u,"[",u.depth(),"] ");//display: 1[0] 2[1] [100,200][2] 3[1]
```

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.

Methods

1. **apply(a,b,c):** *apply a function*
2. **begin():** *initialiaze the iterator with the beginning of the collection*
3. **end():** *return true when the end of the collection is reached*
4. **isvaluetype(string type):** *test the type of the current element*
5. **key():** *return the key of the current element*
6. **nend():** *return true while the end of the collection has not been reached (~end())*
7. **next():** *next element in the collection*
8. **value():** *return the value of the current element*
9. **valuetype():** *return the value type of the current element*
10. **setvalue(value):** *set value to the position pointed by the iterator.*

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,
```

Type date

This type is used to handle dates.

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 **%l**.) (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:

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

Example

```
date mytime;

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

Type time

This type is used to compute timeframes or duration.

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);
```

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.

Methods

1. **eof():** *return* true *when the end of file is reached*
2. **file f(string filename, string moderead):** *open a file according to* moderead*. 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. **write(string s1,string s2,…):** *write strings in the file*

18. **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.*

19. **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();
```

Standard input: stdin

KiF 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.kif, then the content of the piped strings will be displayed with for each line a specific number:

echo “The lady is happy” | KiF stdin.kif.

Type call

This object is used to store a function, 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
```

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.

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 KiF object*
9. **serializestring(object):** *Serialize as an XML document any KiF 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.*

Associated function

The associate function must have the following signature:

```
function xmlnode(xml n, object);
```

It must be declared in the following way:

```
xmldoc mydoc(obj) with xmlnode;
```

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.

Methods

12. **child():** *return the first child node under current node*
13. **child(xml):** *Add an XML node as a child*
14. **content():** *Return the content of a node*
15. **content(string n):** *Change the content of a node.*
16. **delete():** *delete the current internal node.*
17. **line():** *return the line number of the current node*
18. **id():** *return the id of the current node (only with call functions)*
19. **name():** *return the XML node name*
20. **name(string n):** *Change the XML node name*
21. **namespace():** *Return the namespace of the current node as a vector.*
22. **new(string n):** *Create a new internal node.*
23. **next():** *return the next XML node*
24. **next(xml):** *Add an XML node after the current node*
25. **parent():** *return the parent node above current node*
26. **previous():** *return the previous XML node*
27. **previous(xml):** *Add an XML node before the current node*
28. **properties():** *Return the properties of the XML node*
29. **properties(map props):** *Properties are stored in map as attribute/value*

30. **root():** *return the root node of the XML tree*
31. **xmlstring():** *return the XML sub-tree as a string.*
32. **xmltype() :** *return the type of the XML node.*

As a string

Return the XML node name

Example

```
function test(xml n, self nn) {
   map m=n.properties();
   println(n.name(),m,n.content());
}


xmldoc doc with test;

doc.load("resxip.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");
```

Type kif

The type *kif* is used to load a specific KiF program dynamically.

Methods

1. **kif var(string kifpathname):** *Create and load a KiF program*
2. **_loader:** *A* kif *variable (of the type described here), which stores a pointer to the loading program.*
3. **clean():** *this function closes a session and clean it.*

4. **debugfunction(debfunction,object):** *Enable a debug mode in which debfunction is called before each instruction (see below)*

5. **exposed():** *Return a vector of exported function by the KiF program. Each element is a function.*
6. **int firstinstruction=compile(string code):** *this function compiles a piece of KiF code and returns the first instruction of that code. More than one* compile *can be called before running the code.*

7. **load(string kifpathname):** *Load a KiF program*
8. **name():** *Return the pathname of the grm file*
9. **open():** *this function opens a session and return a session handler.*

10. **run(int firstinstruction):** *this function runs a piece of KiF code that has been compiled with* compile *from the first instruction in a given session.*

11. **runasthread(int firstinstruction):** *this function runs a piece of KiF code that has been compiled with* compile *from the first instruction in a given session but as a thread.*

12. **runend():** *this function returns true if the code has been fully executed.*

Executing External Functions

The functions available in the KiF file can be called through a *kif* variable.

Example

In our program test.kif, we implement the function: Read

test.kif

```
function Read(string s) {
        //we then call a function in test.
        _loader.End("From ‘call’ with love");
```

```
        return(s+"_toto");
}
```

call.kif

In our calling program, we first load test.kif, then we execute *Read*

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

//we implement a local function, which will be called from test through _loader…
function End(string s) {
        println("We come back:",s);
}
```

As a string

Return the pathname of the *KiF* file

As a Boolean

Return *true* if a *KiF* file has been loaded.

Cross-reading

If you need to access some variables or some functions from a program that has already been loaded elsewhere, then KiF guaranteed that as long as the path is the same, the other loadings will point to the same version that was loaded in the first place.

So if *prg1.kif* loads *prg2.kif* and *prg2.kif* loads *prg3.kif*, which loads *prg1.kif* again. The second *prg1.kif* will point to same memory space as the initial loading of that program.

Another way to refer to another program in memory is to add a method, which instantiates the current *kif* onto another one. We use *this* in this case to refer to the *calling* program. The only constraint is that this *method* should implement a *kif* parameter.

prg1.kif

```
kif kf('c:\prg2.kif'); //we load a grammar, implementing a set method
kf.set(this); //we set our local kif into prg2.kif.
```

prg2.kif

```
kif caller;

//This function will be called from prg2 to instantiate caller with
//a reference to the loader. prg2 can then call functions implemented in //prg1.kif

function set(kif c) {
   caller=c;
}
```

_loader

In this example, *caller* and *_loader* will share the same value. Furthermore, in the case of a *kif* program loaded in a grammar, which has been loaded in another *kif* program, *_loader* will point to that initial *kif* program. Thus, it is possible to implement *callback* method in *kif*, which will be called from a *kif* program used in a grammar environment.

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 operator

KiF also provides a specific function to load into the current KiF space a KiF program. This operator can placed anywhere in the code, and the variables and functions present in that program are then available to the next instructions.

loadin can take a second parameter, which is a Boolean expression, whose execution decides on whether the file is loaded.

Example

```
loadin('c:\files\prgm.kif'); //this program contains vccc, a vector
println(vccc); //which is now available in the current program
loadin(_args[0],"tst" in _args[0]); //this program loads if its name contains tst
```

Session: open, clean, compile, run

A session is a local compiling and execution of some KiF code from strings. A session needs the following four instructions:

Example

The example below reads a series of instructions and runs them. In this example, we compile our code, line by line, but it is not mandatory. We could compile the whole code in one single string.

```
kif session;
session.open();                 //First we open a session
int first=session.compile("int i=10;"); //we compile the first line, which returns the position of the first instruction
session.compile("int j=10;");     //we add new lines. We do not need any new first instruction of course
sessions.compile("println(i,j);"); //and a print
session.run(first);// we run from the first instruction provided by the first _compile: it prints: 10 10
int second=session.compile("j=i+10;"); //we can add new lines again...
session.compile("println(i,j);"); //no need to run the previous lines
```

```
session.run(second);//we run again, from a new instruction position
//it prints: 10 20

session.clean(); //end of the session
```

setdebugfunction(infos,obj)

It is possible to have a specific function being executed before any execution of any KiF instruction. *debugfunction* can be used at this effect. The first parameter is a function whose signature is the following:

```
function infodebug(string fname,string topname,int ln,string label,self
obj);
```

with:

- fname: *the current file name from which the code is executed*
- topname: *the current function name which is executed*
- ln*: the current line in the file*
- label*: if a label has been declared before the current line*
- obj: *the data object that was passed a parameter to setdebugfunction*

_variables()

It is possible to access the variables from within *infos* with this method, which returns all the active variables, at the level of the current instruction, within the debugging code.

IMPORTANT

The debugging function (here *infodebug)* must return either *true* or *false*. When it returns *false* the current code execution is stopped.

debugclear()

This method clears the debug function mode to revert to a normal execution environment.

Specific instructions

KiF provides all the necessary operations to handle all sorts of algorithms: *if, else, elif, switch, for, while.*

if—elif—else

if (booleanexpression) {}

elif (booleanexpression) {}

…

else {}

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
}
```

for operators.

There are different flavours of “for” in KiF. 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.

You can use the method “*inkey*” to get the current index value in the loop. “*inkey*” uses as input, the variable in which we are looping.

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,inkey(v)); //we also display the current position in the v while looping…
```

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);
```

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;
}
```

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);
```

Evaluation: eval(string code);

This function can evaluate and run some KiF *code* on the fly. The result of the evaluation is returned according to what was evaluated.

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.

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 or three 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.
If only the container is supplied, then the default separator is the carriage return.

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

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:\xip\test\test.txt',true);
println err("This string is now stored in file: test.txt");
iorestate(o,true); //back to normal
```

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)*

Random number: random()

KiF 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.

Example:

```
float rd=random(); // value between 0 and 99
rd=random(999); //value between 0 and 999
```

Keystroke: getc()

KiF also provides a specific function *getc(),* which is used to return a keystroke. *getc()* returns the character code if the input variable is a *int* or a *float* or the character itself if the input variable is a string. To transform a *int* into its encoding character, use *chr().* Below is an example of a small program that reads a string as *get().*

Example

```
int c;
string message;
while (message!=".") {
        print(">");
        c=0;
        message="";
        while (c!=13 && c!=10) {
                c=getc();
                if (c!=13 && c!=10)
                        message+=c.chr();
                print(c.chr());
        }
        println();
        println("End:",message);
}
```

use(OS,library)

use loads dynamic compatible library in a KiF 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 KIFLIBS 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","kifsqlite");***

Persistent Variables: ithrough,fthrough,sthrough,vthrough

These types of variable only make sense when you are using a Graphical User Interface, in which you can run your programs over and over again.

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);
```

…

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.

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");
```

Operator *in*

This operator is quite complex to handle, this is why we have a specific section dedicated to it. In the previous description, we have already described some possible utilization of that operator with files, vectors, maps or strings. We will now see how it can be extended to encompass also frames.

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.

Operator

The operator *in* can be used with a comparison function. This function is introduced with the *with* operator. It is called at each step in the object recursive analysis. This function compares the value with each element of the object and returns true or false accordingly.

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 function*

//In this function, *i* will always be instantiated with the value on the left of *in.*

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

if (3 in vect with compare)
        print("OK");

lst=3 in vect with compare;

//In our example above, i=4…
```

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);
        }
}
```

Operator *on*

This operator introduces some functional programming into the language. The “on” operator is used to apply a function to a container. It is quite similar to the *apply* method, which is already associated with containers with a big difference, “on” expressions can be embedded.

There are basically two “on” operators, which translate into two different sorts of function.
These two “on” operators are not really different in form, but quite different in their output. Again, as it often the case in KiF, the selection of the “right” operator depends on the context.

Contexts

The contexts in this case are either the recipient variables, or the specific context in which the operator is called. We assume in the following examples, that f0, f1are functions and c a container.

```
int v= f0 on c; //in this case, we want the execution of the "on" to be an integer value

v= 10+(f0 on c); //This case is a little different from the one above, but the context again is an integer operation

ivector iv= f1 on c; // This case is very different, we expect the output to be a container

v= f0 on (f1 on c); // this case is the most complex. f1 on c0 is supposed to return a container and f0 on this output container will return a value
```

As we can see on these examples above, the context requires the “on” operator to return either a value or another container.

Basically, the rule is very simple, if the context expects a value as in the two first examples, then the function should return a value.
If the context requires a container then the function should return a container. If the context is underspecified, then the function will return also a container as it is the case in *f1 on c.*
In all cases, the choice of the function relies on the user decision.

Two sort of functions

Actually, the two functions, which will be used according to the context, have almost the same signature.

The container function

The container function should be called in a container context.

```
function fcontainer(iterator it);
function fcontainer(iterator it, self s); //to carry data throughout the process
function fcontainer(iterator it, self s=10); //with an initial value
```

The first parameter is always an iterator, which can be used to get access to both the index and the value. Note that an iterator can also *modify the local value with the method: "set".*
The second parameter is optional. It should be a "self" type if you want your function to carry its values through the whole process. If you do not provide the initial value, then type of "s" will be similar to the first element of your container.

Important: The value returns by this function will *be stored in a container.*

The value function

```
function fcontainer(iterator it);
function fcontainer(iterator it, self s); //to carry data throughout the process
function fcontainer(iterator it, self s=10); //with initial value
```

The value functions are very similar to the above description. However, these functions should not return any values.

Important: *The "s" variable will be the one that will be returned as the final result of your function, otherwise the function returns* ***true****.*

Lambda functions

KiF also accepts *lambda* functions with the "on" operator. A *lambda* function is a function which is declared "on site", in lieu of a more global declaration. There is little difference between a *lambda* declaration and a function declaration, however the body of the function and its name should be declared before the call to the "on" operator. A *lambda function* is declared with the keyword *lambda* followed by the arguments and the function body:

```
lambda(iterator it,self b=1) {your code…} on container;
```

NB: *Lambdas can only be used with the "****on****" operator.*

return(empty)

When these functions return "empty", then the whole analysis is stopped.

Example: Simple Container Function

In this example, we will simply return a container, in which the values from the initial container have all been incremented with 1.

```
//Our initial container
imap iv={"a":1,"b":2,"c":4,"d":8,"e":16,"f":32};
imap result;
```

```
//Our increment function, which returns the current value + 1
function increment(iterator it) {
    return(it.value()+1); //Each value will be stored in our recipient container…
}

//Our call to increment on iv...
//Since result is an imap, each value returned by increment
//will be stored in the container
result=increment on iv;
```

The result is: {'a':2,'b':3,'c':5,'d':9,'e':17,'f':33}
Note that we have kept the same keys…

Example: A More complex Container Function

```
//Our initial container
imap iv={"a":1,"b":2,"c":4,"d":8,"e":16,"f":32};
imap result;

//Our multiply function, which multiply each value with the previous ones
//The initial value for "b" is 1...
//"b" is kept alive across the whole process
function fmultiply(iterator it,self b=1) {
    b*=it.value();
    return(b);
}

//Our call to fmultiply on iv...
//Since result is an imap, each value returned by fmultiply
//will be stored in the container
result=fmultiply on iv;
```

The result is: {'a':1,'b':2,'c':8,'d':64,'e':1024,'f':32768}

Example: Value Function

```
//Our initial container
imap iv={"a":1,"b":2,"c":4,"d":8,"e":16,"f":32};
int result; //Our result is an integer…

//Our multiply function, which multiply each value with the previous ones
//The initial value for "b" is 1...
//"b" is kept alive across the whole process
function fmultiply(iterator it,self b=1) {
    b*=it.value();
}

//Our call to fmultiply on iv...
result=fmultiply on iv;
```

The result is: 32768

Example: embedded calls

```
//Our initial container
imap iv={"a":1,"b":2,"c":4,"d":8,"e":16,"f":32};
int result; //Our result is an integer…

//Our multiply function, which multiply each value with the previous ones
//The initial value for "b" is 1...
//"b" is kept alive across the whole process
function fmultiply(iterator it,self b=1) {
    b*=it.value();
}
```

```
//Our increment function, which returns the current value + 1
function increment(iterator it) {
    return(it.value()+1); //Each value will store in our recipient container…
}

//Our call to fmultiply on iv...
result=fmultiply on (increment on iv);
```

The result is: 151470

Example: a lambda function

```
//An initial string
string s="12:1-14:2";
//We apply our lambda to the split on the "-"
vector svr=lambda(iterator it) {return(it.value().split(":"));} on s.split("-");
```

The result is: [['12','1'],['14','2']]

Functional Language: *à la* Haskell

KiF 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 KiF some of the expressiveness power of the Haskell language with a specific focus on a selected range of functions. We do not pretend that KiF behaves as a full Haskell compiler, but it supplies some of interesting aspects of this language.
In the rest of this chapter, we will still use “Haskell” as a way to refer to the subset of the language that was integrated into KiF, even though we are conscious that we did not go very far into the very fabric of that language.

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 Haskell. These operators are also available in KiF, but their interest really lies in the way they enrich the Haskell 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].

1. **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]...

2. **Infinite ranges**

Haskell 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…

Two new operators: &&& and ::

These two operators are used to concatenate a list of elements together or to add an element to a vector.

1. **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 Haskell. Since this operator was already defined in KiF, we modified it into "&&&".

2. **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*

Basics

Declaring a Haskell-like instruction

All Haskell instructions in KiF should be declared between "<@..@>", which the internal KiF compiler utilizes to detect a Haskell formula.
If the sequence is not ambiguous, you can simply write: "<..>".

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 Haskell 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…

N.B.

If you need to use the binary operator "|", then the expression should be in parentheses, as this operator might be ambiguous with the "|" separator (see below).

- < (12|3) > returns 15…

Ambiguous Sequences

There are cases where the use of "<..>" is ambiguous as in:

<join <x | x <- v, *x.size()> " "*>

Where, we want to apply this operation to object with a *size different from 0.*

The parser will complained that it did not find the actual closing bracket, as it is interpreted here as a comparison between *x.size()>" "*.

To avoid this problem, simply replace the closing “>” with “@>”.

< join <x | x <- v, *x.size() @> “ "* >

Iteration

The Haskell language provides a very convenient and efficient way to represent lists. In KiF, these lists are implemented into “vectors”, which could then be exchanged between the different structures.

The most basic Haskell 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]

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…**

Important Note

If you want to return an operation as a value, as exemplified in the above example, you must use parentheses: (x+y)…

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]]**

Iteration bis

KiF already provides an *iterator* type, which can be used in these Haskell expressions:

- **<x.value() | iterator x=v, x.value()!=0>**

Example:

```
<x.key() | iterator x=[-10..20], x.value()%2==1>
```

yields: **[11,13,15,17,19,21,23,25,27,29]**

Iterations in maps

KiF also provides a specific way to iterate among maps, which in this case is quite different from what is usually available in Haskell implementations.

KiF 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 Haskell 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'}**

Declaring a local variable

There are different ways to declare local variables in a Haskell expression.

1. **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>

However, it is also possible to return a value through an “in” expression. In that case, the different declarations will share one single “let”.

- <let a=10,b=20 in a+b>

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>**

2. **where operator**

The “where” operator is used to declare global variables. It is placed at the end of a Haskell 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 always end with “;”.

- **<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 Haskell 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
```

Functions

The Haskell language also provides a way to declare functions. These functions are equivalent to a KiF function in which all arguments are of type “self”.

Declaration

A function is declared in the following way:

- <name(a1,a2…) : Haskell expression>;

They can be called from a KiF program with: name(p1,p2…).

Examples:

```
<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]**

Actual implementation

In fact, when you declare a Haskell function, KiF 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.

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.

Multiple declarations

It is actually possible to declare a function in more than one Haskell 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 and creates 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…

Haskell 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']`

Note also the ";" at the end of each line…

Calling a function in a Haskell expression

You can call any function or method, either Haskell or KiF in a Haskell 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 Haskell call`

```
//We define a KiF 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 Haskell flavor**

Note that *any method or function* can be called from within a Haskell expression as long as it matches the element type.

- **<adding(v) : <sum v>>;**

There is no actual difference between calling a function in a KiF way or using a Haskell 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
```

Operations

Haskell provides a set of specific functions, which can only be used in Haskell expressions. These functions are used to apply functions or methods to a list or to filter specific values out. Other functions 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 -> …)

1. **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)

2. **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)

3. With a lambda

A lambda in Haskell 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]**

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 KiF function or as a Haskell 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]**

<and (condition) list>

This instruction returns *true* if *condition* is verified for each element.

- **<and (odd) . [1,3..11]> yields *true***

or (condition) list>

This instruction returns *true* if *condition* is verified for at least one element.

- **<or (odd) . [1,2..10]> yields *true***

<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.

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 : ∑ v1 * v2
<dot(v1,v2) : sum . zipWith (*) v1 v2>;
//The norm implementation : √(∑ x²)
<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 vaue 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));
```

Variables *with functions:* Associate Functions

The "*with"* operator has already been described before. It is also possible to associate a function to a variable or a frame with this operator. This *associate* function will be automatically called whenever the value of the variable *changes*. Such a function requires two parameters, the first one being the value of the variable before and the second one, the new value for this variable. This association is done once for all when the variable is declared. If the variable belongs to a frame, then it is possible to implement a three parameters function, in which case, the first parameter is the frame to which this variable belongs.

With operator for basic types

The "with" operator can be used with any types of variables such as *string, int, float* or *containers.* The signature of the associate function depends of course on the type of variable, but also whether it is in a frame or not.

Basic types: int, string, float etc.

For these basic types, the associate function is called whenever the variable is modified. The function must have two parameters of the same type as the variable.

Example

```
function wfonc(string s1,string s2) {
    println(s1,s2);
}

//Note that the intialisation should take place after the function declaration
string s with wfonc="jj";

s="jkjk";
```

Containers: vector, map etc.

For these types, the function is called when a value is added to the container with a "=" sign. In that case, the function will display the key and the value that were either inserted or modified into the container.

Example

```
function wcontainer(int v,string vv) {
    println(v,vv);
}
```

```
vector vx with wcontainer;

vx[0]="jj";


map m with autre;

m[10]="kskdlsad";
```

Frames

In the case of a variable declared into a frame, the function signature will be the same as above, but the first parameter should be a frame.

Example

```
function calllocal(testcallwith tx,int before,int after) {
    println("LOCAL",tx,before,after);
}


//We declare a variable in the frame with an associate function
frame testcallwith {
    //a local associate function
    int i with calllocal=10;
}
```

Example with a variable

```
//We implement a trigger function
function react(string key,string value) {
        println("dico",key,value);
}

//which we associate with dico
map dico with react;

//whenever dico is modified, the "react" function is called
dico["e"]=2;
```

Example in a frame

```
//We first implement our anode frame
frame anode {
        int i;
        function string() {
                return(i);
        }
        function seti(int j) {
                println("J=",i,j);
                i+=j;

        }
}

//First part of our connexe frame
frame connexe {
        vector v;
}
```

```
//a trigger function, the first parameter is a connexe element
function transconnexe(connexe f,int before,int after) {
        iterator it=f.v;
        //we iterate on each element of our vector
        for (it.begin();it.nend();it.next())
                it.value().seti(before+after);
}

//Then the rest of our frame
frame connexe {
        //i is associated with our new trigger function
        int i with transconnexe;

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

        function addanode(int j) {
                anode xn;
                xn.i=j+10;
                v.push(xn);
        }
}

//The transconnexe function will be automatically called, each time i is modified.
connexe c(10); //It will be called here
c1.i=100; //and here
```

Synchronization

KiF offers a simple way to put threads in a wait state. The process is very simple to put in place. KiF provides different functions at this effect:

1. ***string s=wait(string,string,string)***: this instruction stops the execution of the current function and put it in a wait state. It waits for one of the strings it is waiting on to be “cast”. It then returns as a result the string which awoke it.

2. ***cast(string):*** this instruction releases the execution of all threads *waiting* on *string*.

3. ***cast():*** this instruction releases all threads, whatever their *string* state.

4. ***kill(string):*** this instruction kills all pendant threads *waiting* on the same string.

5. ***kill():*** this instruction kills all pendant threads, whatever their *string* state.

6. ***waiting():*** this instruction returns a vector of all pending threads.

7. ***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.

8. ***unlock(string s)***: this instruction deletes a *lock* to enable other threads to get access to the content of a function.

9. ***synchronous:*** this function should be associated to a variable declaration with *with.*

10. ***waitonfalse(var):*** this function put a thread in a wait state until the value of *var* is set to *false. Var* must be declared as *synchronous.*

11. ***waitonjoined():*** this function waits for threads launched within the current thread to terminate. These threads must be declared with the flag *join.*

- **Example:**

```
thread  compte(int i) {
        int j=10;
        i+=1;
        i+=1;
```

```
        i+=1;
        j+=i;
        println("j=",j);
        wait("Here");
        println("I=",i,j);
}

thread comptebis(int i) {
        int j=10;
        i+=1;
        i+=1;
        i+=1;
        j+=i;
        println("bis j=",j);
        wait("Here");
        println("bis I=",i,j);
}

compte(5);
comptebis(10);
println("We come back");
cast("Here");
```

Execution

If we execute the program above, KiF will display in the following order:

j= 18
bis j= 23
We come back
I= 8 18
bis I= 13 23

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.

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.

Semaphores: waitonfalse and synchronous.

If the above functions are useful in a multi-threaded context, there are not enough in some cases. KiF provides two other 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.*

…with synchronous

This first function must be associated with a variable at the declaration level, through a *with*. The role of this function is to be triggered by any modification of this variable in order to detect whether this variable *returns false.* Any variable, including frame variables, can be associated with a *synchronous* function.

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 as synchronous
//Important: its initial value must be different from 0
int stopby with synchronous=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, KiF 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.*

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");
```

Inference engine

KiF embeds an inference engine, which can be freely mixed with regular KiF 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 KiF 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.

Types

KiF 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 name*
2. size(): *return the number of arguments*
3. _trace(bool): *activate or deactivate the trace for this predicate, when it is the calling predicate.*

_predicatemode(mode,compute_final,compute_inter)

This function is used to set a specific mode to deal with weights. *mode* can take different values, that can be combined together, with the | operator:

- pred_none: this is the standard node, where weights are not taken into account…
- pred_weight: this mode applies a Veterbi algorithm to choose among the highest weights first. When a solution is found, it acts as a cut and returns the solution with the highest probability.
- pred_normalize: this mode normalizes the weights so that the sum of all weights is 1.
- pred_randomize: this mode defines a random path among predicates. If used in conjunction with pred_weight, then it stops at the first solution.
- pred_weightstack: this mode pushes on a stack the different weights encountered during the resolution. This mode can be associated with two functions:
    - function compute_final(fvector fv): This function computes a final weight which is associated to the final predicate. This function is provided by the user. If this function is not supplied, then the system computes a product of each weight normalized with the size of the stack.
        - N.B. Important, this function receives a *fvector* as input, which has been built as in a stack along a predicate resolution path. *If you modify some of these values, they will be stored back into their original predicates, when the system is back-tracking.*
    - function compute_inter(fvector fv): This function must return *true* or *false*. If *false* is returned then the function acts as a *fail.*

Example:

```
predicate sentence;

term s,np,vp,d,n,v;

function calc(fvector v) {
    return(v.product()/v.size());
}

int decompte;
```

```
function inter(fvector v) {
    decompte++;
    if (decompte>10)
        return(false);
    return(true);
}

sentence._trace(true);
_predicatemode(pred_weightstack,calc,inter);

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<0.1>(d("the")) --> ["the"].
det<0.4>(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);
```

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*

- KiF 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])]
```

- KiF also provides an associative map, which is implemented as a KiF 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
```

Clauses

A clause is defined as follow:

predicate<weight>(arg1,arg2…,argn) :- pred(arg…),pred(arg,…), etc. ;

The "<weight>" is optional. It is used while analyzing to push some predicates up. When weights are provided, they defined a deterministic search.

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

KiF 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

KiF 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

Kif 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.*

assertdb(pred(…),db)

This function asserts (inserts) a predicate into a SQLite database. *Db* should be a *sqlite* object (see *store)*

retract(pred(…))

This function removes a predicate from the knowledge base. Note that this function *can only be used within a clause declaration.*

retractdb(pred(…),db)

This function removes a predicate from the database. Db should be a *sqlite* object (see *remove*).

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.*

ponder(pred)

This function is used to force a ponderation on a fact. If no weight is associated to the predicate, then its value will be 1.

Example: **ponder(parent<0.1>(“Tess”))**

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.

Result:

If we run our above example, we obtain:

Parent: parent('John','Mary')
Parent: parent('John','Peter')

Persistence

Persistence is done with the help of a Sqlite database, which should be declared as follow (see *sqlite* type for more information):

Example:

```
sqlite db;
db.open('mydb');
```

Declaration

To make a fact persistent, you must declare it within the database, with the method: *predicate*.

```
db.predicate(name,arity);
```

The *arity* is the predicate number of arguments. This function creates a table, whose name is "*name_arity*", and with as many columns named [P0,P1..PN] as indicated by *arity*.

Operators

Fact can be made persistent with the following operators:

- **store(db)**: *stores a predicate in the **sqlite database db.***
- **get(db)**: *queries the database for a given predicate*
- **remove(db)**: removes a predicate from the database.
- **assertdb(predicate,db):** *store a predicate in the sqlite database db. Assertdb is much more versatile than store(db), which requires that all information be known.*
- **retractdb(predicate,db):** *removes a predicate from the database.*

Example 1

```
female('mary') :- store(db); //we store this predicate in our database.
```

DCG

KiF 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);
```

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.

Common mistakes with KiF 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)
```

Some 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')]
```

Ancestor again but with a database

```
predicate ancestor,parent,female,test,truc;

sqlite db;

//Our database declaration
db.open('persitent.db');

//We declare our predicates
 db.predicate("female",1);
 db.predicate("parent",2);

//Transaction mode
db.begin();

//Storing our predicates
female("stephanie") :- store(db).
female("liliane") :- store(db).
female("jeanne") :- store(db).
female("mary") :- store(db).
female("francoise") :- store(db).

parent("george",10.3234) :- store(db).
parent("george",100) :- store(db).
parent("george","sam") :- store(db).
parent("george","andy") :- store(db).
parent("andy","mary") :- store(db).

db.commit();

//Now we tell the system where to look for the predicate values
parent(?X,?Y) :- get(db).
female(?X) :- get(db).

//But here: no change...
ancestor(?X,?X) :- true.
ancestor(?X,?Z) :- parent(?X,?Y),ancestor(?Y,?Z).


test(?X,?Q) :- ancestor(?X,?Q), female(?Q).

vector v=test("george",?Z);

println(v);

db.close();
```

Ancestor (last), with assertdb instead of store

```
predicate ancestor,parent,female,test;

sqlite db;
```

```
//Our database declaration
db.open('persitent.db');

string err;

//We declare our predicates in the database, with their arity...
try {
    db.predicate("female",1);
    db.predicate("parent",2);
}
catch(err) {
    println(err);
}

//We need these two instructions to store our values in the database...
addingfemale(?X) :- assertdb(female(?X),db).
adding(?X,?Y) :- assertdb(parent(?X,?Y),db).

//Now we tell the system where to look for the predicate values
//THESE ARE TWO important lines, since we give here the link between the database
and the predicates...
parent(?X,?Y) :- get(db).
female(?X) :- get(db).

//An ancestestor has a a parent, who is 'self a parent...
ancestor(?X,?X) :- true.
ancestor(?X,?Z) :- parent(?X,?Y),ancestor(?Y,?Z).


//We are looking for a an ancestor, whose descendant is a female
test(?X,?Q) :- ancestor(?X,?Q), female(?Q).
test._trace(true);

//We add the following siblings to our database...
//Note the ";" at the end of the line, to trigger the evaluation of our predicates.
//Here, the main difference with the previous example is obvious, since we can define
whatever
//values we want on the fly.
db.begin();
addingfemale("stephanie");
addingfemale("liliane");
addingfemale("jeanne");
addingfemale("mary");
addingfemale("francoise");
adding("george","sam");
adding("george","andy");
adding("andy","mary");
db.commit();

//Looking for a female with a specific parent...
vector v=test("george",?Z);
println(v);

//Testing whether "mary" is a female.
println(female("mary"));

//Looking for all parents...
v=parent(?X,?Y);
println(v);

db.close();
```

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(th
e),nom(dog))))),sentence(['the','cat','eats','the','bat'],[],P(SN(dét(the),nom(cat)),SV(verbe(eats),S
N(dét(the),nom(bat))))),sentence(['the','cat','eats','a','cat'],[],P(SN(dét(the),nom(cat)),SV(verbe(ea
ts),SN(dét(a),nom(cat))))),sentence(['the','cat','eats','a','dog'],[],P(SN(dét(the),nom(cat)),SV(verb
e(eats),SN(dét(a),nom(dog))))),sentence(['the','cat','eats','a','bat'],[],P(SN(dét(the),nom(cat)),SV(
verbe(eats),SN(dét(a),nom(bat))))),sentence(['the','dog','eats','the','cat'],[],P(SN(dét(the),nom(do
g)),SV(verbe(eats),SN(dét(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','the','bat'],[],P(S
N(dét(the),nom(dog)),SV(verbe(eats),SN(dét(the),nom(bat))))),sentence(['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),nom(dog)),SV(verbe(eats),SN(dét(a),nom(dog))))),sentence(['the','dog','ea
ts','a','bat'],[],P(SN(dét(the),nom(dog)),SV(verbe(eats),SN(dét(a),nom(bat))))),sentence(['the','bat
','eats','the','cat'],[],P(SN(dét(the),nom(bat)),SV(verbe(eats),SN(dét(the),nom(cat))))),sentence(['t
he','bat','eats','the','dog'],[],P(SN(dét(the),nom(bat)),SV(verbe(eats),SN(dét(the),nom(dog))))),se
ntence(['the','bat','eats','the','bat'],[],P(SN(dét(the),nom(bat)),SV(verbe(eats),SN(dét(the),nom(ba
t))))),sentence(['the','bat','eats','a','cat'],[],P(SN(dét(the),nom(bat)),SV(verbe(eats),SN(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(dé
t(a),nom(bat))))),sentence(['a','cat','eats','the','cat'],[],P(SN(dét(a),nom(cat)),SV(verbe(eats),SN(d
ét(the),nom(cat))))),sentence(['a','cat','eats','the','dog'],[],P(SN(dét(a),nom(cat)),SV(verbe(eats),S
N(dét(the),nom(dog))))),sentence(['a','cat','eats','the','bat'],[],P(SN(dét(a),nom(cat)),SV(verbe(eat
s),SN(dét(the),nom(bat))))),sentence(['a','cat','eats','a','cat'],[],P(SN(dét(a),nom(cat)),SV(verbe(ea
ts),SN(dét(a),nom(cat))))),sentence(['a','cat','eats','a','dog'],[],P(SN(dét(a),nom(cat)),SV(verbe(eat
s),SN(dét(a),nom(dog))))),sentence(['a','cat','eats','a','bat'],[],P(SN(dét(a),nom(cat)),SV(verbe(eat
```

s),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','dog','eats','the','dog'],[],P(SN(dét(a),nom(dog)),SV(verbe(eats),SN(dét(the),nom(dog))))),sentence(['a','dog','eats','the','bat'],[],P(SN(dét(a),nom(dog)),SV(verbe(eats),SN(dét(the),nom(bat))))),sentence(['a','dog','eats','a','cat'],[],P(SN(dét(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))))),sentence(['a','dog','eats','a','bat'],[],P(SN(dét(a),nom(dog)),SV(verbe(eats),SN(dét(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),nom(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),nom(bat)),SV(verbe(eats),SN(dét(a),nom(dog))))),sentence(['a','bat','eats','a','bat'],[],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();
```

kifsys

KiF provides a library, which offers some system functionalities such as, reading the content of a directory into a vector or executing a system command.

To use this library: use('kifsys');
kifsys also exports the type: *sys* together with a *sys* variable: *kifsys*, which can be used to execute system commands.

Methods

1. **command(string s,string outputfile)**: *execute*[1] *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

1

e. *info["directory"]: true* if the path is a directory

f. *info["pathname"]:* the real pathname

9. **realpath(string path):** *return the actual path for a given relative path.*

Example

```
use('kifsys');
//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=kifsys.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=kifsys.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
				kifsys.command(cmd);
			}
		}
	}
}

//We call this function to copy from one directory to another
cp('C:\src','W:\src');
```

kifsocket

kifsocket is a specific library which exports the type *socket* to handle socket interactions between a client and a server.

To use this library: use('kifsocket');

Methods

Client Side

1. **close():** *close the socket*
2. **close():** *close the socket*
3. **close(clientid):** *close the communication with clientid.*
4. **connect(string hostname,int port):** *connect a client to a specific host on a specific port.*
5. **createserver(int port,int nbclients):** *create a server on the local host with a specific port.*
6. **createserver(string hostname,int port,int nbclients):** *create a server on a host with a specific port.*
7. **get() :** *get one character from a socket*
8. **get(int clientid) :** *get one character from a socket with clientid.*
9. **getframe(string name):** *return a frame object* remote *handle of name* name.
10. **getfunction(string name):** *return a function* remote *handle of name* name.
11. **gethostname():** *return the current host name. The socket does not need to be activated to get this information.*
12. **read():** *read a KiF object on the socket*
13. **read(clientid):** *read a KiF object on the socket with clientid*
14. **receive(int nb):** *read nb characters from a socket*
15. **receive(int clientid,int nb):** *read nb characters from the socket with clientid*

16. **run(int client,string stopstring):** *put the server in run mode. Server can now accept Remote Method Invocation (RMI) mode.*
17. **send(int clientid,string s):** *write a simple string on the socket with clientid*
18. **send(string s):** *write a simple string on the socket*

Server Side

19. **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.*
20. **wait():** *the server wait for a client to connect. It returns the client identifier, which will be used to communicate with the client.*
21. **write(clientid,o1,o2...):** *write KiF objects on the socket with clientid.*
22. **write(o1,o2...):** *write KiF 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();
```

A server to parse sentences

The example, which is described below, shows how a XIP server can be put in place in KiF. This server receives sentences from the clients, which are parsed. The result of the parse is returned as a string.

The server itself

```
//We load the socket library
use('kifsocket');
parser french;
//we load a grammar
french.load('C:\FRENCH\french_eerqi.grm');

//This function will be used to launch the parse of a sentence.
function parsing(string sx) {
        string res=french.parse(sx);
        return(res);
}

//We read on a socket the sentence from the client...
thread parse(int num,socket s) {
        string sx;
        string res;
        string mes;
        while (true) {
                try {
                        //we read our sentence
                        sx=s.read(num);
                        if (sx!="") {
                                //the we parse it
                                res=parsing(sx);
                                //we return the result to the client
                                s.write(num,res);
                        }
                }
                catch(mes) {
                        //in case of error, we terminate the thread…
                        s.close(num);
                        return;
                }

        }
}

socket s;
//We create our server on the port 2021, for 20 potential clients
```

```
s.createserver(2021,20);
println("Port: 2021");
int n;
while (true) {
	//we wait for a client to connect
	n=s.wait();
	//we use this socket id to parse the sentence...
	parse(n,s);
}
```

The client

```
use('kifsocket');

socket s;
//we connect to our server, the first argument is the port
s.connect(s.gethostname(),_args[0]);

//We set a timeout of 5s, in order to avoid infinite socket reading...
s.settimeout(5);

file f;
//We open our file, which is the second argument of our program
f.openread(_args[1]);
string ph;
string res;
int compte=0;


function parse(string p) {
	int e;
	//We display the chunk number
	println("Example:",compte);
	compte+=1;
	try {
		//we send our sentences to the server
		s.write(p);
		//which returns the result as a string
		res=s.read();
	}
	catch(e) {
		//If a error occurs
		println("Erreur:",e);
		if (e==830) {
			//830 is a TIME OUT error
			println("Time out");
			//we close the current socket
			s.close();
			//we reconnect to the server
			s.connect(s.gethostname(),_args[0]);
			//we try to write our sentence once again
			s.write(p);
			//and to read the result
			res=s.read();
		}
	}
	println(res);
}

//we read each string from the file
for (ph in f) {
	ph=ph.trim();
	if (ph!="")
```

```
                parse(ph);
}
```

remote

kifsocket offers a second type: remote, which is used to run methods and functions on a remote server. The server must launch a *run* on the server side to handle execution requests from the client.

Get frames and functions

The *client* on the other hand, must request frames and functions through two socket methods: *getframe* and *getfunction* in order to launch these executions.
By default, any objects and any functions can be requested by the client, which limits the necessity to specific declarations on the server side.
These two methods return a *remote* object, which can be used to run any functions or methods.

Private

However, if you want to hide within your server, frames and functions, you must declare them as *private*, which will prevent any attempts from the client to reach specific functions or frame variables.

String or vector

When a *remote* object is handled as a vector, it returns the list of available functions on the server. In the case of a *getfunction*, the list will be limited to one single element. However, in the case of a *getframe,* all available functions (non *private)* will be stored.

Run parameter

The *string* parameter for the *run* function is used to allow the client to shut down the server, by sending a simple *write* with that specific string. To prevent a client from shutting down a server, one possibility is to launch a *run* with the empty string as a parameter.

Server side

The code below implements a frame and a function. This server implements a function: *create*, which creates a new *test* object as a global variable (via an object broker), and a frame object *toto*, which is accessible from the client.

The server can be stopped with a specific string, here: *stop.*

```
//A simple server of objects...
```

```
//First we declare a simple frame
frame test {
        int i;

        //This function should not be called on the client side
        private function _initial(int x) {
                i=x;
        }

        //This function cannot be called on the client side
        private function Set(int j) {
                i=j;
    }

        function Value() {
                return(i);
        }

        function Compute(int j) {
                return(i*j);
        }
}

//The method creates a new frame test object, with the name: n
function create(string name,int i) {
        test t(i);
        _KIFMAIN[name]=t;
}

//We have our specific frame object, which any client can ask for
test toto(100);

//This object is declared as “private” and cannot be requested by the client…
private test mineDoNotTouch(-1);

//We create our server
socket s;

println(s.gethostname());
//On port: 2012
s.createserver(2012,10);

//we wait for a client to connect
int client=s.wait();
//The stop string is here, well: "stop"
//We wait for the client to execute functions on the server side...
s.run(client,"stop");
//If our server receives the "stop" string, it stops.
s.close();
println("Stopped");
```

Client side

The client side implements a call to the *toto* frame object, with an execution of each of the methods exposed by this frame. It also calls *create* to create a new object on the server side, which is in its turn executed on the server side. Finally, it writes the *stop* string, which kills the server.

```
//We connect to our local server.
socket s;
```

```
string name=s.gethostname();
s.connect(name,2012);
//The we declare a remote object: r
remote r;

//We get from the server a handle on toto, which is implemented on the server side
r=s.getframe("toto");

//We display the available methods from the frame description of toto, which have
been returned by the server.
println("Functions:",r);

println("Values of toto:",r.Value(),r.Compute(345));

//We then fetch a function: create
r=s.getfunction("create");

//Which we can use to execute the object created on the server side...
r.create("titi",123);

//we can now fetch a titi object, that was created thanks to the above instruction
r=s.getframe("titi");
//And we display its values
println("Valeur de titi:",r.Value(),r.Compute(345));

//We kill the server, by sending the killing string.
s.write("stop");
s.close();
```

kifsqlite

KiF 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. *kifsqlite* exports the type: sqlite

To use this library: use('kifsqlite');

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
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("Deja cree");
}

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.nend();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();
```

LMDB (Lightning Memory-Mapped Database)

LMDB is another database manager, but of the form key/value. LMDB has been developed by Symas for the OpenLDAP project (see http://symas.com/mdb/ for more information).

We have mapped the *LMDB* manager onto KiF maps, but with some specific enhancement over these maps.

Methods

1. **open(string pathname,int size,int nbdbs,int flags,int fixedsize):** *open a database. See below for an explanation of the* flags. *If "fixedsize" is 0, then records can have different sizes. If you set "fixedsize" with a specific number, then* keys *and* values *will have this "fixed" size to store their specific values, making the whole database access faster, but less versatile.*
2. **begin():** *start the transaction mode*
3. **create(string table,int flags):** *create a table*
4. **commit():** *commit transactions*
5. **selfcommit(bool):** *commit transaction is automatically done when using 'var[key]=value'*
6. **put(string key,string value,int flags):** *store a value*
7. **delete(string key):** *delete a key*
8. **cursormode(int cursorflag):** *set the cursor mode access*
9. **get(string key,int cursorflag):** *Get a value with a key, with the size sz along a cursorflag... Use an iterator to get values, without a call to 'begin'*
10. **close():** *close a session*
11. **select(string table):** *switch to another table*

Opening Flags

1. MDB_NODUPDATA - *enter the new key/data pair only if it does not already appear in the database. This flag may only be specified*

if the database was opened with MDB_DUPSORT. The function will return MDB_KEYEXIST if the key/data pair already appears in the database.

2. MDB_NOOVERWRITE - *enter the new key/data pair only if the key does not already appear in the database. The function will return MDB_KEYEXIST if the key already appears in the database, even if the database supports duplicates (#MDB_DUPSORT). The data parameter will be set to point to the existing item.*

3. MDB_RESERVE - *reserve space for data of the given size, but don't copy the given data. Instead, return a pointer to the reserved space, which the caller can fill in later. This saves an extra memcpy if the data is being generated later.*

4. MDB_APPEND - append the given key/data pair to the end of the database. No key comparisons are performed. This option allows fast bulk loading when keys are already known to be in the correct order. Loading unsorted keys with this flag will cause data corruption.

5. MDB_APPENDDUP - *as above, but for sorted dup data.*

Cursor mode Flags

Here is the list of cursor mode flags (also the flags that are used in 'get'). Each corresponds to a KiF constant variable.

1. MDB_CURRENT - *overwrite the data of the key/data pair to which the cursor refers with the specified data item. The key parameter is ignored (to be used with* get*)*
2. MDB_FIRST: *Position at first key/data item*
3. MDB_FIRST_DUP*: Position at first data item of current key (Only for MDB_DUPSORT)*
4. MDB_GET_BOTH*: Position at key/data pair (Only for MDB_DUPSORT)*
5. MDB_GET_BOTH_RANGE*: position at key, nearest data (Only for MDB_DUPSORT)*
6. MDB_GET_CURRENT*: Return key/data at current cursor position*
7. MDB_GET_MULTIPLE*: Return all the duplicate data items at the current cursor position (Only for MDB_DUPFIXED)*
8. MDB_LAST: *Position at last key/data item*
9. MDB_LAST_DUP*: Position at last data item of current key (Only for MDB_DUPSORT)*
10. MDB_NEXT *: Position at next data item*
11. MDB_NEXT_DUP*: Position at next data item of current key (Only for MDB_DUPSORT)*
12. MDB_NEXT_MULTIPLE: *Return all duplicate data items at the next cursor position (Only for MDB_DUPFIXED)*
13. MDB_NEXT_NODUP*: Position at first data item of next key. (Only for MDB_DUPSORT)*

14. MDB_PREV: *Position at previous data item*
15. MDB_PREV_DUP*: Position at previous data item of current key (Only for MDB_DUPSORT)*
16. MDB_PREV_NODUP*: Position at last data item of previous key (Only for MDB_DUPSORT)*
17. MDB_SET*: Position at specified key*
18. MDB_SET_KEY*: Position at specified key, return key + data*
19. MDB_SET_RANGE*: Position at first key greater than or equal to specified key.*

Errors

LMDB returns the following error values:

1. MDB_KEYEXIST: key/data pair already exists
2. MDB_NOTFOUND: key/data pair not found (EOF)
3. MDB_PAGE_NOTFOUND: Requested page not found - this usually indicates corruption
4. MDB_CORRUPTED: Located page was wrong type
5. MDB_PANIC: Update of meta page failed, probably I/O error
6. MDB_VERSION_MISMATCH: Environment version mismatch
7. MDB_INVALID: File is not a valid MDB file
8. MDB_MAP_FULL: Environment mapsize reached
9. MDB_DBS_FULL: Environment maxdbs reached
10. MDB_READERS_FULL: Environment maxreaders reached
11. MDB_TLS_FULL: Too many TLS keys in use - Windows only
12. MDB_TXN_FULL: Txn has too many dirty pages
13. MDB_CURSOR_FULL: Cursor stack too deep - internal error
14. MDB_PAGE_FULL: Page has not enough space - internal error
15. MDB_MAP_RESIZED: Database contents grew beyond environment mapsize
16. MDB_INCOMPATIBLE: Database flags changed or would change

Iterators

It is possible to use iterators with *lmdb* variables. Actually, iterators can even be triggered with a get and continued with a *next*. See the example below to see how to start this sort of iteration.

Example

```
//We need to load it first (it is not part of KiF binary)
```

```
use('kiflmdb');

//We declare an lmdb variable
lmdb lm;
//We open a directory, which must exist otherwise it is an error.... The size here is 10000000
lm.open('C:\XIP\Test\lmdb\mydb',1000000,0,0);
//we create a table
lm.create("",0);

//This instruction is used so that each association lm[key]=v is automatically stored
//If you deactivate it, you will need to commit, which of course is usually the best solution as selfcommit is much slower....

//lm.selfcommit(true);

//We store our values in the table....
int i;
string akey;
for (i=0;i<100;i++)  {
    akey="K"+i;
    lm.put(akey,i,0);
}
//We have two ways to store things, we can use put or a map
lm["KK"]=-29;
//In all cases, we need to commit, unless for the above instruction we have set: selfcommit,
//which is not a good idea usually.
lm.commit();

//We look for values, note that you can again use a map instruction...
println(lm["K3"],lm["KK"]);

//Or you can use 'get', with the flag: MDB_SET_KEY
lm.get("K2",MDB_SET_KEY);

//You can also use iterator among values
iterator it=lm;
//No need for begin, as the above 'get' has already triggered a loop among values... Hence, the initialization which has been replaced with a “true” in the for loop.

for (true;it;it++) {
    //We only display the values we are interested in
    if (it.key()!="K2")
        break;
    println("Key:",it.key());
    println("Value:",it.value());
}

//we close our database...
lm.close();

println("Ok");
```

Result

```
['3'] ['-29']
Key: K2
Value: 2
Ok
```

LIBLINEAR

KiF 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.

Methods

1. **cleandata()**: *clean internal data*
2. **crossvalidation()**: *Relaunch the cross validation with new parameters. The result is a fmap.*
3. **load(string inputdata,smap options,string outputfilename)**: *load your training data with some options.* outputfilename *is optional. It will be used to store the final model if provided.*
4. **loadmodel(string filename)**: *Load your model. A model can also automatically be loaded with a constructor.*
5. **options(smap actions)**: *Set the training options (see below)*
6. **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.*
7. **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.*
8. **savemodel(string outputfilename):** *save your model in a file*
9. **trainingset(fvector labels,vector data):** *create your training set out of a treemapif vector.*

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:

```
smap 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 *ifreemap*, 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.

Training example

```
//we load the library
use("kifliblinear");

string trainFile = "output.dat";

//we declare a liblinear variable
liblinear train;
```

```
//We set the options
smap 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("kifliblinear");

//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);
```

LIBSVM

KiF provides also an encapsulation of the *libsvm* library (see http://www.csie.ntu.edu.tw/~cjlin/libsvm/ for more information). This library is used to implement classifiers and exposes the following methods. It uses the same data schema as *liblinear* above.

Methods

1. **cleandata()**: *clean internal data*
2. **crossvalidation()**: *Relaunch the cross validation with new parameters. The result is a fmap.*
3. **load(string inputdata,smap options,string outputfilename)**: *load your training data with some options.* outputfilename *is optional. It will be used to store the final model if provided.*
4. **loadmodel(string filename)**: *Load your model. A model can also automatically be loaded with a constructor.*
5. **options(smap actions)**: *Set the training options (see below)*
6. **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.*
7. **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.*
8. **savemodel(string outputfilename):** *save your model in a file*
9. **scale(string inputfile,map options):** *Scale files along options:*
   i. **'l':** *lower_value*
   ii. **'u':** *upper_value*
   iii. **'y':** *y_lower_value*
   iv. **'Y':** *y_upper_value*
   v. **'r':** *restore_file_name*
   vi. **'s':** *save_file_name.*

Training options

The training options should be supplied as a map with the following keys and values:

's': svm_type : set type of SVM (default 0)
0 -- C-SVC (multi-class classification)
1 -- nu-SVC (multi-class classification)
2 -- one-class SVM

```
	3 -- epsilon-SVR	(regression)
	4 -- nu-SVR		(regression)

't': kernel_type : set type of kernel function (default 2)
	0 -- linear: u'*v
	1 -- polynomial: (gamma*u'*v + coef0)^degree
	2 -- radial basis function: exp(-gamma*|u-v|^2)
	3 -- sigmoid: tanh(gamma*u'*v + coef0)
	4 -- precomputed kernel (kernel values in training_set_file)

'd': degree : set degree in kernel function (default 3)
'g': gamma : set gamma in kernel function (default 1/num_features)
'r': coef0 : set coef0 in kernel function (default 0)
'c': cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR (default 1)
'n': nu : set the parameter nu of nu-SVC, one-class SVM, and nu-SVR (default
0.5)
'p': epsilon : set the epsilon in loss function of epsilon-SVR (default 0.1)
'm': cachesize : set cache memory size in MB (default 100)
'e': epsilon : set tolerance of termination criterion (default 0.001)
'h': shrinking : whether to use the shrinking heuristics, 0 or 1 (default 1)
'b': probability_estimates : whether to train a SVC or SVR model for probability
estimates, 0 or 1 (default 0)
'wi': weight : set the parameter C of class i to weight*C, for C-SVC (default 1)
'v': n: n-fold cross validation mode
```

kifcrfsuite

kifcrfsuite is an encapsulation of the crfsuite (see www.chokkan.org/software/crfsuite/ for more details) software.

crfsuite provides an 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:

www.chokkan.org/software/crfsuite/manual.html

kifcrfsuite exposes the following methods:

Methods

1. **close():** *close a model.*
2. **learn(svector args):** *Learn a model from an input file. The list of arguments should match the arguments as described in the crfsuite documentation.*
3. **load(svector args):** *Load a model that will be used for tagging. The options should match the list of options described in the crfsuite manual.*
4. **separators(string separator,string endoftagging):** *setting how the fields will be separated. By default,* separator *is "#" and* endoftagging *is* "$".
5. **tag(svector tokens):** *apply the model to a vector of tokens. The result is a vector of tokens.*
6. **tagfile(string filename,string outputfile):** *apply the model to an input file and store the results in an outputfile. If* outputfile *is not provided, then the result is return as a vector.*

File Formats

For a better description of the file formats, see the documentation. Basically, a CRF file is composed (both for training and prediction) of different columns separated with tab characters. The first column is the tag that will be learnt our output by our model. The other columns contains rules, which define how the different features for each tag will be produced.

Example:

VERB	w[0]=Judging	w[1]=from	pos[0]=VERB	__BOS__			
PREP	w[-1]=Judging	w[0]=from	w[1]=previous	pos[-1]=VERB	pos[0]=PREP	pos[1]=ADJ	
ADJ	w[-1]=from	w[0]=previous	w[1]=posts	pos[-1]=PREP	pos[0]=ADJ	pos[1]=NOUN	
NOUN	w[-1]=previous	w[0]=posts	w[1]=this	pos[-1]=ADJ	pos[0]=NOUN	pos[1]=PRON	
PRON	w[-1]=posts	w[0]=this	w[1]=used	pos[-1]=NOUN	pos[0]=PRON	pos[1]=VERB	
VERB	w[-1]=this	w[0]=used	w[1]=to	pos[-1]=PRON	pos[0]=VERB	pos[1]=PREP	

```
PREP    w[-1]=used      w[0]=to  w[1]=be pos[-1]=VERB    pos[0]=PREP     pos[1]=VERB
VERB    w[-1]=to w[0]=be w[1]=a   pos[-1]=PREP    pos[0]=VERB     pos[1]=DET
DET     w[-1]=be        w[0]=a   w[1]=good        pos[-1]=VERB    pos[0]=DET      pos[1]=NADJ
NADJ    w[-1]=a  w[0]=good       w[1]=place       pos[-1]=DET     pos[0]=NADJ     pos[1]=NOUN
NOUN    w[-1]=good      w[0]=place       w[1]=,   pos[-1]=NADJ    pos[0]=NOUN     pos[1]=PUNCT
PUNCT   w[-1]=place     w[0]=,   w[1]=but         pos[-1]=NOUN    pos[0]=PUNCT    pos[1]=CONJ
CONJ     w[-1]=,  w[0]=but       w[1]=not         pos[-1]=PUNCT  pos[0]=CONJ     pos[1]=ADV
ADV     w[-1]=but       w[0]=not         w[1]=any         pos[-1]=CONJ    pos[0]=ADV      pos[1]=ADV
ADV     w[-1]=not       w[0]=any         w[1]=longer      pos[-1]=ADV     pos[0]=ADV      pos[1]=ADV
ADV     w[-1]=any       w[0]=longer      w[1]=.   pos[-1]=ADV     pos[0]=ADV      pos[1]=PUNCT
PUNCT   w[-1]=longer    w[0]=.   pos[-1]=ADV     pos[0]=PUNCT    __EOS__
```

Examples

Training

```
use("kifcrfsuite");
crfsuite c;

//Our options: we apply a 10 fold cross-validation, the result is stored in CRF.model
//the training file is training.crf
string options="-g10 -x "+" -m "+_current+"CRF.model "+_current+"training.crf";
svector voptions=options.split();

//we apply our training...
c.learn(voptions);
```

Applying

```
use("kifcrfsuite");
crfsuite c;
string options="-i -p -m "+_current+"CRF.model";
svector voptions=options.split();
//we load our model...
c.load(voptions);
//we apply the our model to a file
vector v=c.tagfile(_current+"example.crf");
//here the output is in a file
c.tagfile(_current+" example.crf",_current+"output.txt");
//In this case, we load the file and store its content as a vector.
file f(_current+"example.crf","r");
svector vs=f.read();
vs=<x.trim() | x <- vs>;
f.close();
//we apply our model to this vector of tokens.
v=c.tag(vs);
println(v);
c.close();
```

kifwapiti

kifwapiti is an encapsulation of the wapiti library, which is available on:

http://wapiti.limsi.fr .

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

kifwapiti exposes the following methods.

Methods

7. **loadxip(string crfmodel, string brown,string mkcls):** *Loading the CRF files for tweet tagging.*
8. **loadmodel(string crfmodel):** *Loading a CRF model.*
9. **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.*
10. **train():** *Launch training. Requires options to have been set in advance.*
11. **label(vector words):** *Launch labelling for a vector of tokens. Returns a vector of labels for each token.*
12. **analyze(svector words):** *Analyze a sequence of words and returns the predicted tags, according to loadxip files.*
13. **lasterror():** *Return the last error, that did occur.*

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**
- **–l | --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**

XIP

Note: *loadxip and analyze* do not require any options. They load the necessary file to run the taggers that have been trained over tweets. If you need to train a “pure” CRF system, do not use these two methods.

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('kifwapiti');

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();
```

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('kifwapiti');


//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','ORGANISATIO
N_i','null','null','null','null','null','null','null','null','null']
```

kifword2vec

KiF 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.*

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.*

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("kifword2vec");

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();
```

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("kifword2vec");

word2vec wrd;

//we load the model that was obtained through training
wrd.loadmodel("output.bin");
svector v=[“word”];

fmap res=wrd.distance(v);
```

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("kifword2vec");

word2vec 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...]
```

Predicates

You can use this property to unify two variables in a predicate, if the distance between two *w2vector* is superior to their threshold.

Example:

```
use("kifword2vec");

word2vec model;
model.loadmodel("mymodel.bin");


afact("Accident","Fukishima ","tsunami ","accident ").
afact("Incident","Fukushima ","earthquake","incident ").


evaluation(?X) :- afact("Accident",_,?X,_).
comp(?X,?Y) :- evaluation(?X),afact("Incident",_,?Y,_).

//we then call our predicates with two w2vector, built on the fly
//Their threshold is set to 0.3. If we compare a w2vector to a string, then
//the distance is automatically computed with this string and returns TRUE
//if it is superior to the w2vect threshold of 0.3.
//Here, model[“damage”:0.3] will be compared against “accident” and
//“incident”.
vector vres=comp(model["earthquake":0.3],model["damage":0.3]);
```

Fast Light ToolKit library (GUI)

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 KiF 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 KiF library. However, the available methods are enough to build simple but powerful interfaces.

The name of the library is: kifltk.dll (.so)

Note

a) We have linked KiF with FLTK 1.3.0.

b) The associate function methodology has been extended to most graphical objects.

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
```

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];

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();
```

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* when available.

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. **pushclip(int x,int y,int w, int h):** Insert a clip region, with the following coordinates
44. **rectangle(int x,int y,int w, int h, int c)**: *Draw a rectangle with optional color c*
45. **rectanglefill(int x,int y,int w, int h, int c)**: *Fill a rectangle with optional color c*
46. **redraw()**: *Redraw the window*
47. **resizable(object)**: *make the object resizable*
48. **rgbcolor(int color)**: *return a vector of the color decomposition of into RGB components*
49. **rgbcolor(int r,int g,int b)**: *return the* int *corresponding to the combination of RGB components.*
50. **rgbcolor(vector rgb)**: *return the* int *corresponding to the combination of RGB components, which are stored in a vector.*
51. **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.*
52. **run()**: *Launch the GUI*
53. **size():** *return a 4 values vector of the window size*

54. **size(int x,int y,int w,int h):** *resize the window*
55. **sizerange(int minw,int minh, int maxw,int maxh)**: *define range in which the size of the window can evolve*
56. **textsize(string l)**: *Return a map with w and h as key to denote width and height of the string in pixels*
57. **unlock():** *FLTK unlock*

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

```
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

```
//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

Kifltk 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.
KiF 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

Kiflitk 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.

```
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

Kifltk 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.

```
//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 KiF, 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

```
//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

```
//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

```
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

```
//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.

```
//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

```
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

```
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();
```

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

```
//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();
```

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	Nothing selected when items are clicked.
FL_TREE_SELECT_SINGLE	Single item selected when item is clicked (default)
FL_TREE_SELECT_MULTI	Multiple 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

```
//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

```
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();
```

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

```
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);
```

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*

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
```

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

```
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);
```

wchoice

kifltk 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

```
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();
```

wtable

kifltk 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

```
window w;

function callback_table(table 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();
```

editor

Kifltk 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. m*atchcase is optional.*
3. **annotateregexip(string reg,string keystyle):** *Each string matching the xip regular expression reg is assigned the style keystyle.*
4. **append(string s):** *append a string at the end of the editor text*
5. **byteposition(int pos)**: *convert a character position into a byte position (especially useful in UTF8 strings)*
6. **charposition(int pos)**: *convert a byte position into a character position (especially useful in UTF8 strings)*
7. **color(int c)**: *set or return the text color*
8. **copy():** *copy selected text to clipboard*
9. **copy(string s):** *copy string s to clipboard*
10. **count(string s,int bg,int mx):** *count the number of occurrences of s, between bg and mx.*
11. **create(int x,int y,int w,int h,string label)**: *Create an editor*

12. **cursor():***return the current position of the cursor in byte increments.*
13. **cursor(int i):** *move the cursor to the i^{th} bytes.*
14. **cursorchar():***return the current position of the cursor in character increments.*
15. **cursorstyle(int style):** *set the cursor shape (see below)*
16. **cut():** *cut selected text to clipboard*
17. **delete():** *delete selected text*
18. **e[a:b]:** *Extract characters between a and b*
19. **e[a]**: *Extract character from the editor at position a*
20. **find(string s,int i):** *find a string in the editor text, starting at position* i.
21. **font(string s)**: *set or return the text font*
22. **fontsize(int c)**: *set or return the text font size*
23. **getstyle(int start,int end):** *Return the style for each character of a chunk of text as a vector*.
24. **gotoline(int l,bool highlight):** *goto line* l *and highlight it if true.*
25. **highlight():** *return 1 or 0 if there is highlighted text in the editor. In the context of a string, returns the highlighted string*
26. **highlight(int start,int end):** *highlight the characters between* start *and* end*.*
27. **insert(string s,int pos):** *insert a string at position pos*
28. **line()**: *return the current line number or the line text itself*
29. **line(int pos)**: *Return the line corresponding to pos or the line text itself. This line should be visible.*
30. **linebounds()***: return a vector with the start position and end position of the current line in bytes increments.*
31. **linebounds(int pos***): return a vector with the start position and end position of the line at pos in bytes increments.*
32. **lineboundschar()***: return a vector with the start position and end position of the current line in character increments.*
33. **lineboundschar(int pos***): return a vector with the start position and end position of the line at pos in bytes increments.*
34. **load(string filename):** *load the content of a file into the editor.*

35. **onhscroll(function f, object o)**: *set the callback when scrolling horizontally (see below for an example)*
36. **onkey(int action,function f, object o)**: *set the callback when scrolling vertically (see below for an example)*
37. **onmouse(int action,function f, object o)**: *set the callback when handling the mouse*
38. **onvscroll(function f, object o)**: *set the callback when scrolling vertically (see below for an example)*
39. **paste():** *paste selected text to clipboard*
40. **rfind(string s,int i):** *find a string in the editor text, starting at position* I, backward.
41. **save(string filename):** *save the content of the editor into a file.*
42. **selection()**: *return the selected text in the editor*
43. **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)*
44. **unhighlight():** *remove highlighting*
45. **value(string v)**: *return the text in the editor or initialize the editor*
46. **word(int pos):** *return the word at position pos*
47. **wrap(boolean w):** *wrap the text within the editor window if w is true.*

Cursor shape

KiF 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 ],
          };
```

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

```
//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

```
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();
```

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

wprogress

kifltk 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:

```
window w;

wprogress c;

thread progressing () {
    for (int i in <0,100,1>) {
        for (int j in <0,100000,1>) {}
        c.value(i);
    }
    println("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();
```

wcounter

kifltk 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

7. **bounds(float x,float y):** *defines the counter boundary*
8. **create(int x,int y,int w,int h,int alignment, string label):** *Create a counter*
9. **font(int s):** *set or return the text font*
10. **lstep(double):** *define the large counter step*
11. **step(double):** *define the counter step*
12. **steps(double):** *define the counter steps, normal and large.*
13. **textcolor(string code|int code):** *Set the color of the text*
14. **textsize(string l):** *Return a map with w and h as key to denote width and height of the string in pixels*
15. **type(bool normal):** *if 'true' then normal counter or simple counter*
16. **value(float):** *define the value for the counter or return its value*

Example:

```
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();
```

slider

kifltk 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

17. **align(int align)**: *define the slider alignement*
18. **bounds(int x,int y)**: *defines the slider boundaries*
19. **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)*
20. **resize(object)**: *make the object resizable*
21. **step(int)**: *define the slider step*
22. **type(int x)**: *Value slider type (see below for the list of slider types)*
23. **value()**:*return the slider value*
24. **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.

```
//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();
```

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.

```
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.

```
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

```
//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);
```

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

```
//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:\XIP',"*", FL_DIR_SINGLE,"Choose your file");
```

A simpler solution is:

```
filebrowser f;
string value=f.create(".","*.*",FL_DIR_SINGLE,"Open");
println("Value:",value);
```

sound

KiF 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. KiF relies on *libao4, libsndfile-1* and *libmpg123* for decoding and playing.

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

```
sound s;

s.load('C:\XIP\XIP7\sound\Kalimba.mp3');
s.play();
```

You can also load a sound with the following declaration:

```
sound s('C:\XIP\XIP7\sound\Kalimba.mp3');
```

Decoding example

```
//we open a sound file
sound s('C:\XIP\XIP7\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();
```

type curl (WEB Page loading)

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.

This library should be loaded with: *kifcurl* as name.

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.*

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.

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('kifcurl');
```

```
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('kifcurl');

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"]);
```

Type python (kifpython)

The *kifpython* library provides the necessary methods to execute some Python code in KiF. It exposes a new type: *python.*

The base KiF 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 KiF into Python variables
def Iteste(s,v):
  v.append(s)
  v.append(val)
  return v
```

Then we implement our own KiF program, which will call this file (which we suppose to be in the same directory as our KiF program)

```
//We need to use kifpython for our own sake
use("kifpython");

//we need a variable to handle the Python handling
python p;

//we suppose that our Python program is in the same directory as our KiF program
p.setpath(_paths[1]);
//We then import our program
p.import("testpy");

vector v;
```

```
string s="kkk";

//We execute the Python function ltest, 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 KiF vector (from the Python vector)
vector vv=p.execute("ltest",s,v);

println(vv); //output is: ['kkk','here']

p.close(); //we close the session
```

Type wordnet (Based on WordNet 3.0)

The WordNet 3.0 library has been embedded into a KiF library. It can be called from any program with the command:
use("kifwordnet").

This library exposes the following methods:

Methods

6. **findinfo(string word,string pos,string type,string senses,int definitions)**: *Retrieve all senses of a word according to the different flags (see below for a list of type, senses and pos flags)*
   definition *can have the following values:*
   0 *is no definition and no list of POS.*
   1 *adds the list of POS*
   2 *adds the definitions.*
   3 *adds all.*

7. **findsynset(int synset,string pos,int definition,string word)**: *Retrieve all senses associated with a synset. 'word' is optional.*
   definition *can have the following values:*
   0 *is no definition and no list of POS.*
   1 *adds the list of POS*
   2 *adds the definitions.*
   3 *adds all.*

8. **frequency(string word,string pos)**: *Return the 'frequency' of a word as a value between 0..7, 0 being the less frequent*

POS

The list of part of speeches is the following:

ALL_POS,NOUN,VERB,ADV,ADJ,ADJSAT

TYPE

The list of types, which can be used within a *findinfo* method is the following:

ANTPTR, HYPERPTR, HYPOPTR,ENTAILPTR, SIMPTR, ISMEMBERPTR, ISSTUFFPTR, ISPARTPTR, HASMEMBERPTR, HASSTUFFPTR, HASPARTPTR, MERONYM, HOLONYM, CAUSETO, PPLPTR, SEEALSOPTR, PERTPTR, ATTRIBUTE, VERBGROUP, DERIVATION, CLASSIFICATION, CLASS

If you want more information about these flags, please refer to WordNet documentations.

Senses

ALLSENSES, SYNS, FRAMES, COORDS, RELATIVES, HMERONYM, HHOLONYM, WNGREP, OVERVIEW, CLASSIF_START, CLASSIF_USAGE, CLASSIF_REGIONAL, CLASS_START, CLASS_USAGE, CLASS_REGIONAL, INSTANCE, INSTANCES

If you want more information about these flags, please refer to WordNet documentations.

Example

```
use("kifwordnet");

map res;
res=kifwordnet.findinfo("drink","NOUN","HYPERPTR","ALLSENSES",1);
println(res);
```

The result is the following:

```
{
 'synset':7885223,
 'words':['drink'],
 'pos':'n',
 'nextsynset':{
  'synset':748515,
  'words':['drink','drinking','boozing','drunkenness','crapulence'],
  'pos':'n',
  'nextsynset':{
   'synset':7881800,
   'words':['beverage','drink','drinkable','potable'],
   'pos':'n',
   'nextsynset':{
    'synset':9270508,
    'words':['drink'],
    'pos':'n',
    'nextsynset':{
     'synset':839778,
     'words':['swallow','drink','deglutition'],
     'pos':'n',
     'whichword':2,
     'synsets':[838098,1170052,1201856,840057,840189,843494]
    },
    'whichword':1,
    'synsets':[9225146,7075172]
   },
   'whichword':2,
 'synsets':[21265, 14940386, 797113,1170052, 7844042, 7882420,7883251,
7884567, 7890970, 7891095, 7891189, 7891309, 7913180, 7914006, 7914128,
7914271, 7919310, 7921455, 7922764, 7924033, 7925966, 7926785, 7927197,
7929519, 7933274, 7933530, 7936263]
  },
  'whichword':1,

'synsets':[748011,10537,10385,10537,1172275,1171183,1172275,1171183,748834]
,
 },
 'whichword':1,

'synsets':[7578363,1170052,7883860,7883980,7884413,7885937,7912499,7912619,
7912726,7912834,7912933,7913081,7916773,7916872,7916970,7918454,7918601,792
3034,7923297,7923665]
}
```

The result is a complex map, with the following keys:

- *synset* is the synset of the current element (see wordnet documentations for more details)
- *synsets* is a list of other synsets matching the current word.
- *pos* is the current part of speech
- *ppos* is a list of part of speech matching *synsets*, when the option *definition=1 or definition=3* is set.
- *nextsynsets* is another possible instantiation for the current word.
- *whichword* is the current position of the word in the *synsets* list.
- *words* is the list of word strings matching the *synsets* list.
- *definition* is a definition given for the current word, is *definition* is set to 2 or 3.

KiF API

It is possible to execute a KiF program through a C++ API.

KiF provides four functions to this effect, which should be used in the following way:

- **int KifLoad(vector<string>& paths,vector<string>& arguments,bool debugkif)**

This instruction loads a list of KiF programs in the same memory space, in a similar fashion as “–kif”, on the command line.

paths is a vector of pathnames, each corresponding to a KiF program to load

debugkif triggers the debug mode

arguments is a vector of arguments which are passed to the KiF program.

This function returns a *handler* on the KiF program which has been loaded.

- **string KifExecute(int ikif,string name,vector<string>& parameters,ostringstream* os,bool debugkif);**

This function runs a specific KiF function, which returns as a result a string.

1. **ikif** is the handler which has been returned by KifLoad. It identifies a KiF or a list of KiF programs.
2. **name** is the name of the function, which should be executed. This function must have been declared in the KiF programs that have been loaded.
3. **parameters** is a vector of strings, which corresponds to the parameters of the function that will be called. Each parameter should be transformed into a string prior to any call to this function.
4. **os** will store any *print* command executed within the KiF program.
5. **debugkif** triggers the debug mode.

Implementing a KiF library

USE operator

KiF provides a specific function: *use*, which is used to load an external compatible library into a KiF program in order to enrich the language with new types and new capabilities. The *use* function must be placed at the beginning of a program before any other definitions.

Example:

```
use('kifsqlite');


sqlite mydb;
mydb.open('C:\XIP\Test\SampleLogGalateas\test.db');
```

It should be noted that external KiF libraries are compiled as either DLL or dynamic libraries according to the platform and thus obeys the way DLL or dynamic libraries are found and loaded. Hence, on *Unix*, dynamic libraries are found according to the variable content: *LD_LIBRARY_PATH.*

Platform and *use*

In another embodiment, this function can take two parameters. The first one is the platform and the second one the path itself. Kif recognizes three different platforms: *WINDOWS, MACOS, UNIX and UNIX64*, which should be provided as strings.

Example

```
// this library will only be loaded on UNIX platforms.
use('UNIX','/usr/lib/kifsqlite');

// this library will only be loaded on WINDOWS platforms.
use('WINDOWS','c:\dlls\kifsqlite');

// this library will only be loaded on MACOS platforms.
use('MACOS','/usr/lib/kifsqlite');


sqlite mydb;
mydb.open('C:\XIP\Test\SampleLogGalateas\test.db');
```

String Comparison

Use can also be used with three parameters, the two first parameters being tested one against the other. The first parameter should be a *_args* variable, whose value will be compared against the next parameter.

Example

```
// this library will only be loaded if the value of _args[0] is “TOTO”
use(_args[0],'TOTO','/usr/lib/kifsqlite');
```

Template

KiF provides a specific C++ template to implement a KiF compatible library. It is not a *template* in the C++ sense, but rather a C++ file implementing anything that is necessary to add new capabilities to KiF.

Replacement

The first operation consists of duplicating the file: *kiftemplate.cxx* into a new file.

In this copy, you will replace the keyword: *TemplateName* with your own new name. This replacement is done in any editor with the replace command. Replace every single instance of *TemplateName* with your new name.

For the sake of the description, we will replace *TemplateName* with the keyword *Try.*

The new file exposes two classes: KifIteratorTry and KifTry.

TemplateName_KifInitialisationModule

This method is the heart of your library. This is the first function that will be called once your library has been loaded.

Once you have replaced every single instance of *TemplateName* in your file with the keyword *Try,* you will discover in this specific function the following line:

```
Try_type=kifcode->KifAddNewType("Try",CreatekifTryElement);
```

Try_type is a KifType variable, which receives as value the new identifier that will be associated to your new object. The first parameter of KifAddNewType is a string, which will be the type that will be used in your KiF program.

```
Try toto;
```

1. *CreatekifTryElement* is the function that the KiF compiler will called when it finds a *Try* type in a KiF program. This function, which is also implemented in your new file, creates a new object of that new type for the compiler.

2. *MethodInitialisation(“sample”,&KifTry::MethodSample,NBARGS,”Description”);* this method is the most important one. It is

the one that defines a new method that will be exposed through your new object. It needs a name (a simple string) and a KifTry method implementation. NBARGS is a combination of one or many predefined values: P_NONE,P_ONE, P_TWO (see kifbase.h for the whole set) etc…

Add as many *MethodInitialisation* as you wish. Do not forget to implement the function itself.

IMPORTANT

The name of this method, which is the entry point of the library is based on the *library name, together with the keyword:* KifInitialisationModule. It is then important to keep the name of your library in par with your *template name*, otherwise, KiF will not be able to find this entry point.
If your library name is *mylib* then this method must be called:

- *mylib_* KifInitialisationModule(...)

Specific Objects: VERY IMPORTANT

KiF implements specific garbage collectors for the following objects, namely:

- KifString
- KifInteger,
- KifFloat,

For these objects, you MUST NOT USE "*new KifString"* or "*new KifFloat"*, but the following methods, which are exposed in KifCode (see kifbase.h):

```
Exported KifString* Providestring(string& z);
Exported KifString* Providestringraw(string z);
Exported KifInteger* Provideinteger(long val);
Exported KifFloat* Providefloat(double val);
```

KiF, for efficiency reasons, maintains lists of predefined objects, which it actually reuses during execution. Thus, KiF has a list of integers, floats or strings from which it extracts the right objects when needed. If you need to return a *string* object, then call: *kifcode->ProvideString(value).*

KifTry

This class is the main class of your new library. It implements the behavior of your library. It derives from *KifObject* and it expects some methods and some new variables. As a C++ class, you need

to declare your own variables, and to instantiate their initial value in the class constructor.

Your new class will implement something similar to the lines below.

```
class KifTry : public KifBasic {
public:
//These two static objects are used for internal method descriptions
static map<string,KifCallMethod> kifexportedmethods;
static map<string,TryMethod> linkedmethods;

//------------------------------------------------------------------------
//This SECTION is for your specific implementation...
//Your personal variables here...
string s; ← WE ADD our new variable here
//------------------------------------------------------------------------
KifTry(KifCode* kifcode,KifElement* base) : KifBasic(kifcode,base,Try_type) {
        s="";
}
```

We have modified the class by implementing a new field: *s*. You can add as many variables as you want.

KifIteratorTry

If you want your new object to have iterator capabilities, you may implement a class like this one.

- You need to provide a *Begin* method, to initialize your iteration. *Begin* returns the first element of your list.
- You need to provide a *Next* method, to go one step further. This method should return the next element of your list.
- You need to provide a *End* method, which return *kifTRUE* is the end of the list is reached, *kifFALSE* otherwise.
- You need to provide a *IteratorKey*, which returns the key of the current element.
- You need to provide a *IteratorValue,* which returns the value of the current element.

Furthermore, an i*terator* provides a *reverse* Boolean variable to indicate the direction of iteration, which might be of some use if you want to provide a reverse iterator to your object.

Newiterator

Newiterator is the method which is called when an iterator is instantiated. If you do not need any iterator, you can remove both this method and *KifIterator* class. If you need an iterator, then this method is the right one to initialize your iteration.

KifTry: Setvalue.

When an KiF finds an instantiation in a program, this is the method that is always called.

```
Try toto;
toto="abcdef";
```

The *Setvalue* method will be called if a program contains a line like the one above.

```
bool Setvalue(KifElement* kval,KifElement* kindex,KifElement* domain) {
        s=kval->String();
        return true;
}
```

Setvalue receives three parameters:

- kval, which is the value with which to instantiate our current object (here its value will be: abcdef)
- kindex, which is an index if you want to handle your object as a list or as a map. In this case, it will be NULL.
- domain, which is the domain in which our variables are present.

If we have the following line:

```
toto[10]="abcdef";
```

Then kindex will have *10* as a value.

KifTry: Copy

You can provide a copy of your object:

```
KifElement* Copy(KifDomain* kp,KifElement* dom=kifNULL) {
        KifTry* kperso=new KifTry(KifCurrent(),NULL);
    //We initialize kperso with our current value
        kperso->s=s;
        return kperso;
}
```

If you do not want to provide any copy, you can then return *this.*

KifTry: String(), Integer(), Float(), Boolean()

These methods are called for you whenever a *Try* object will be in a string, integer, float or Boolean context.

```
string String() {return s;}
```

KifTry: plus, minus etc…

If you want to provide your class with an interpretation of the following operators: +,-,*,/,%,^,&,|,<<,>> then you might want to overload these methods.

```
KifElement* plus(KifElement* a,KifElement* b) {
        string s1=a->String();
        string s2=b->String();
        s1+=s2;
        return kifcode->Providestring(s1);
}
```

KifTry: less, different, more, same: <,>,=,!=,<=,>=

If you want to compare *Try* object together, then you might consider overloading these methods:

```
KifElement* same(KifElement* a) {
        if (s==a->String())
                return kifTRUE;
        return kifFALSE;
}
```

The element *a* is compared with *this.*

KifTry: Clean and Clear

There is slight difference between Clean and Clear. Clean is called when an element is deleted from the garbage collector. Clear is called when an element is simply reset to its initial value. Hence, the "*reference--"* in Clean.

KifTry: Home made methods.

Of course, the goal of such a library is to implement your own code. A new method should be implemented in two steps.

First, call M*ethodInitialization* with the name you have chosen for your new method and the method that should be called in that case.

```
MethodInitialization("command",&KifSys::MethodCommand,P_ONE,"command(strings): Execute a command");
```

Second, implement your *MethodCommand* in KifTry.

```
KifElement* MethodCommand(KifElement* contextualpattern,
        KifDomain* domain,
        KifCallFunction* callfunc) {
        if (callfunc->Size()!=1) //Not really necessary, as you have constrained to only one parameter your method with P_ONE, but it could be important
          kifcode->Returnerror("MYKIF(800): Missing parameter");

        //0 is the first parameter and so on...
        KifElement* kcmd=callfunc->Evaluate(0,domain);
        if (contextualpattern->type==kifString)
```

```
                s=kcmd->String();
        //you may return any value of course...
        return kifTRUE;
}
```

Parameters

A method implemented in this way requires three parameters:

KifElement* contextualpattern; *This variable is a bit difficult to understand. It stores the context in which the action is taking place. For instance, if this method is called through a string initialization, then contextualpattern type will be kifString as in the example below:*

```
string str=toto.command("my string");
```

str in this context imposes a String context to the whole formula, which is reflected in the value of contextualpattern (which in this case will point to *str* KiF representation).

KifDomain* domain; *this variable defines the frame in which the action is taking place.*

KifCallFunction* callfunc; *this variables points to the function call. It enables the access to the parameters.*

A KifCallFunction exposes two main methods:

- Size(), which returns the number of parameters available
- Evaluate(ipar,dom); which extracts and evaluate the parameter at position ipar.

A method should always return a KifElement of any sorts as output.

Note:

The name *MethodCommand* is only for the sake of the demonstration. You can choose whatever name your want as long as the link between its name and the method itself is implemented through a call to *MethodInitialization*.

Frame Derivation

The main difference between implementing a frame version of your library with the previous one relies on the use of *kiframetemplate.cxx* as a source for your own derivation and the use of two specific methods to handle *frame* variables (*field* hereafter).

Keyword: FrameTemplate

First of all, as for *kiftemplace.cxx*, replace the keyword: *FrameTemplate* with your own new name.

Differences

You will notice some difference with the previous version. First difference, the basic class derives from *KifFrame*, which gives access to some new specific methods.

Second, the presence of the variable: *localDefinition,* which is used to share your specific *frame* implementation.

As for the other definition, you only need to define your own *frame* variables or *fields* and your own methods, with two major differences.

Newfield(string name,KifElement* value);

If you want to enlarge your new *frame* with specific *fields*, you need to declare them through a call to *Newfield* in your class constructor. You will find an example of such a declaration in the *kiframetemplate.cxx* file:

```
KifFrameTemplate(KifCode* kcode,KifElement* base,string& name) : KifFrame(kcode,base,name) {

//Mandatory to update the new methods exposed by this frame
        map<string,KifCallMethod>::iterator it;
        for (it=kifexportedmethods.begin();it!=kifexportedmethods.end();it++)
                declarations[it->first]=&it->second;

//We create our own fields.
//For the sake of our example an integer (i) and a string (s).
        Newfield("i",kcode->Provideinteger(10));
        Newfield("s",kcode->Providestringraw("here"));
}
```

The first parameter is the name of your new *field* in your *frame* implementation and the second one is an object, which defines the type of that new object. It might be anything such as: *KifFloat, KifBoolean, KifTime, KifMap or KifVector*.

This is similar to the following *frame* declaration:

```
frame FrameTemplate {
        int i=10;
        string s="here";
}
```

Getfield(string name,KifDomain* domain);

If you implement your own methods, they will need access to the *fields* of the current *frame* object, which *Getfield* will do for you.

```
        KifElement* vi=Getfield("i",domain);
        KifElement* vs=Getfield("s",domain);
```

For instance, in the *sample* method which has been declared in *kiframetemplate.cxx*, you will see the two above lines, which will return the actual variables attached to these two field names.

String(), Integer() etc.

In a non-frame implementation, the method *String, Integer, Float, Boolean, Size* must be provided by the user. However, in a *frame*, the problem is a bit different. No *String* or *Integer* can be implemented directly. As for a *frame,* where the user must provide the right method to evaluate it as a string or as an integer, in the similar way, a *string* method should be provided by the programmer in the same way as other functions.

```
MethodInitialization("string",&KifFrameTemplate::MethodString);
```

In order to implement a *string* method, the programmer must provide a *MethodString* associated to the keyword: *string.*
An example of such a method is available in *kiframetemplate.cxx*.

```
KifElement* MethodString(KifElement* contextualpattern,
                                KifDomain* domain,
                                KifCallFunction* callfunc) {
//No parameter
  if (callfunc->Size()!=0)
            kifcode->Returnerror("KIF(800): Wrong number of parameters");

  KifElement* vs=Getfield("s",domain);
  return vs;
}
```

Frame implementation

kiframetemplate.cxx contains the code of a declaration equivalent to the following one:

```
frame FrameTemplate {
            int i=10;
            string s="here";

            function sample(int xi,string xs) {
                        i+=xi;
                        s+=xs;
                        return(true);
            }

            function string() {
                        return(s);
            }

}
```

XIP Integration

The next pages describe the specific integration of KiF within the *Xerox Incremental Parser* (XIP). The following instructions can only be accessed through XIP with the API, through the “-kif” and “-kifargs” instructions on the XIP command line and of course through the grammars.

Passing arguments to a KiF program: Specific to XIP

There are different ways of passing arguments to a KiF program within XIP. The easiest way is to use *–kifargs* on the command line.*–kifargs* should be the last command in the list of commands as all the strings after this keyword will be passed as an argument to your KiF program.

```
xip –kif mykif.kif –kifargs first second third
```

The other way to pass arguments is to use a “*kifarg:*” field in a GRM file, the content of the field is then passed as an argument to the program. More than one “*kifarg:*” can be declared at a time.

Important

If you instantiate arguments with both *“-kifargs”* and *“kifarg:”*, then it should be noted that the arguments from the GRM file will be stored *before* the ones on the command line.

Garbage collector (Specific to XIP)

Each KiF object is associated with a reference, which is either incremented or decremented according to its use. When a KiF function is called from a XIP rule, then the garbage collector is automatically called and all objects created during the execution of that function are cleaned and removed from memory. However, the garbage collector can also be called whenever a specific threshold is crossed. By default, the threshold value, which is computed as a number of KiF objects created, is set to 10.000. However, it is possible to reset this value from the command line with *-kifsize* and provides a different threshold value.

```
xip –kifsize 100000  etc…
```

KiF function: garbagesize(value);

KiF also exports a specific function, which should be placed at the beginning of the code to set the garbage size value.

XIP API

The XIP API also exports a specific method: *KifSetSize*, which takes as input the new threshold.

```
KifSetSize(int threshold);
```

In this case, whenever the number of KiF objects created is superior to this *threshold*, the garbage collector is called and memory is freed from unused objects.

KiF from XIP

A KiF function can be called from any XIP rule, anytime. The only constraint is that the KiF function must be declared before its utilization in a XIP grammar. Furthermore, any XIP variable can be used in a KiF function. However, the reverse is not true. A KiF variable is not visible from a XIP rule. Finally, a KiF section is a XIP file *should always be the last section in that file.*

Example

```
//File: test.kif
Variables:
string s="in XIP";

KiF:
function display() {
        print("S=",s,"\n");
}

//File: script.xip
Script:
display();
Run
S="in XIP"
```

Handling XIP variables

A KiF function can modify any XIP variable. It can *only return numerical values*.

Example

```
Variables:
string s="in XIP";
int i;

KiF:
function display() {
        print("S=",s,endl);
        s="value from KIF";
        return(2);
}

Script:
i=display();
print("S again:"+s+":"+i+"\n");
```

RUN

S=in XIP
S again: value from KIF:2

XIP objects

XIP objects, such as syntactic nodes, dependencies or generation nodes, can be passed to a KiF function in a transparent manner.

Example with XIP nodes

KiF:

```
function display(node n) {
        print("NODE POS="+n+"\n");
}
```

Script:

```
|Noun#1| {
        display(#1);
}
```

```
RUN
NODE POS: NOUN
```

Example with dependencies

KiF:

```
function display(dependency d) {
        print("Dependency="+d+"\n");
}
```

Script:

```
|Noun#1|
   if (subj$1(#2,#1)) {
        display($1);
}
```

```
RUN
Dependency=SUBJ
```

In a IF or a WHERE

It is also possible to use KiF function in a *if* or in a *where.* The function should return then a numerical value, where *false* is 0 and *true* is anything else.

KiF:

```
function testpos(node n) {
//We need to force the STRING interpretation of n…
   if ("noun"==n)
      return(true);
  return(false);
}
```

Script:

```
|Noun#1|    if (testpos(#1))
   DEP(#1);
```

Important

If KiF is case-sensitive, XIP is not. This might create some issues as for XIP the two functions *Display* and *display* will be a single function. If you implement KiF functions that might be called from a XIP rule, avoid playing on uppercase or lowercase characters to distinguish between different functions.

XIP KiF API

The following C++ instruction describes how KiF programs can be loaded or executed from within the XIP API.

- **int XipKifLoad(vector<string>& paths,vector<string>& arguments,bool debugkif)**

This instruction loads a list of KiF programs in the same memory space, in a similar fashion as “–kif”, on the command line. However, in this case, the KiF programs are loaded within an empty XIP parser object, which enables functionalities such as FST compiling or Callback functions. Furthermore, the initialization of a XIP object also enables the passing of XIP specific parameters to the KiF programs.

1. **paths** is a vector of pathnames, each corresponding to a KiF program to load
2. **debugkif** triggers the debug mode
3. **arguments** is a vector of arguments which are passed to the KiF program.

This function returns a *handler* on the GlobalParseur object.

- **string XipKifExecute(int ipar,string name,vector<string>& parameters,ostringstream* os,bool debugkif);**

This function runs a specific KiF function, which returns as a result a string. However, in this case, we use the handler from the *XipKifLoad* instruction.

1. **ikif** is the handler which has been returned by XipKifLoad. It identifies a KiF or a list of KiF programs.
2. **name** is the name of the function, which should be executed. This function must have been declared in the KiF programs that have been loaded.
3. **parameters** is a vector of strings, which corresponds to the parameters of the function that will be called. Every parameter should be transformed into a string prior to any call to this function.
4. **os** will store any *print* command executed within the KiF program.
5. **debugkif** triggers the debug mode.

Callback functions

If a KiF program loads a grammar through a *parser* object, it is possible through the API to catch on the fly the different analyses which are being produced.

In this case, the programmer must first load the KiF program through a *XipKifLoad* instruction, in order to set a specific callback function through a *XipSetCallBack*. The execution should then be done *through a XipKifExecute instruction.*

C++ program

```
void mycall(int ipar, XipResult* xip,void* data) {…}

//First we load our KiF program
vector<string> paths;
paths.push_back('C:\Test\parse.kif');
vector<string> arguments;
arguments.push_back("C:\Test\French\french.grm");
int ipar=XipKifLoad(paths,arguments,false);

//Then we set our callback function
XipSetCallBack(ipar,mycall,NULL);

ostringstream os;
arguments.clear();
arguments.push_back("La dame mange une glace.");

//We can then execute the function that is exported by our KiF program
string kifres=XipKifExecute(ipar,"parsing",arguments,&os,false);
```

KiF program: parse.kif

```
parser p;

//we load our grammar from within KiF
//args[1] corresponds to: C:\Test\French\french.grm
p.load(args[0]);

//s=La dame mange une glace.
thread parsing(string s) {
        string r;
        //we parse our function
        r=p.parse(s);
        //kifres //we parse our function
        //will receive r as a result (converted as a string).
        return(r);
}
```

The callback function: *mycall* will be automatically passed to the grammar *french.grm,* which will then enable programmers to have access to the XipResult objects generated for that specific parse.

Type node: Syntactic XIP node

When a KiF function is called from a XIP rule, the node parameters #1,#2 etc. can be passed to that KiF function as *node.*

Methods

1. **applied()**: *return the list of rule indexes that applied to a given input as an ivector.*
2. **child():** *return the first child node under current node*
3. **feature(string att):** *return the value of an attribute*
4. **feature(string att,string val):** *test the value of an attribute*
5. **features():** *Return a map a results, in which attributes as stored as keys, and values as...values*
6. **features(smap m):** *Store the values in the map in the XIP node (or dependency), where the maps keys should be the attributes.*
7. **last():** *return the last child node under current node*
8. **leftoffset():** *return the left offset*
9. **leftoffsetchar():** *return the left character offset*
10. **lemma():** *return the lemma string. It should be noted, that if the recipient variable is a vector and the element is ambiguous, then a vector of lemmas can be returned as for pos above.*
11. **lemma(string s):** *replace the lemma of the current node with s.*
12. **lemma(string lem,string pos,smap feats):** *add a new "reading" to a node.* "lem" *is the new lemma,* "pos" *is the part of speech and "*feats" *a feature structure.*
13. **name():** *return the part of speech*
14. **next():** *return the next node after current node. Can also return the next lexical node in the sentence when the structure is flat.*
15. **sister():** *return the next node. If the structure is flat, then* sister() *returns null.*
16. **nodeinfos():** *return the node infos value*

17. **nodeinfos(string v):** *set the node infos value*

18. **number():** *return the ID of current node*

19. **offset(int left,int right):** *left and right receive the offsets*

20. **offsetchar(int left,int right):** *left and right receive the character offsets*

21. **parent():** *return the parent node above current node*

22. **pos():** *return the part of speech. It should be noted, that if the recipient variable is a vector and the element is ambiguous, then a vector of pos can be returned (the same applied to name above).*

23. **previous():** *return the previous node after current node*

24. **readings():** *return a vector of all possible readings for a given word.*

25. **removefeature(string att):** *remove a feature from a node.*

26. **rightoffset():** *return the right offset*

27. **rightoffsetchar():** *return the right character offset*

28. **righttokenoffset():** *return the right token offset*

29. **rulenumber():** *return the rule number, which applied to create this node.*

30. **sentence()**: *return the string spanned by the node out of the buffer.*

31. **setfeature(smap m):** *feature assignment to the node from a map, where the keys are XIP attributes.*

32. **setfeature(string att, string val):** *feature assignment to the node*

33. **surface():** *return the surface string*

34. **surface(string s):** *replace the surface string of the current node with s.*

35. **tokenoffset(int left,int right):** *left and right receive the token offset*

36. **xmlnode():** *return the XML node associated with this node (TOKENIZE mode)*

As a string

Return the part of speech

As an integer or a float

Return the node id.

Example

```
//XIP rule
|Noun#1| if (testkif(#1))…

//KiF code
Function testkif(node n) {
   print("N=",n,"\n");          //display: NOUN
   map features;
   n.features(features);
   vector v=n.lemma(); //more than one value can be returned
   return(true);
}
```

Type dependency

This type corresponds to the $1,$2 etc... dependency variables in XIP.

Methods

1. **feature(string att):** *return the value of an attribute*
2. **feature(string att,string val):** *test the value of an attribute*
3. **features(map feats):** *Features are stored in map as attribute/value*
4. **name():** *return the dependency name*
5. **parameters():** *return a vector of* node *variable*
6. **pop():** *remove the last element on top of the dependency's stack*
7. **pop(int i):** *remove the i^{th} element from the dependency's stack*
8. **push(string s):** *push s on the dependency's stack*
9. **rulenumber():** *return a vector of the rule numbers, which applied to create this dependency.*
10. **setfeature(string att, string val):** *feature assignment to the dependency*
11. **stack(vector v):** *return the dependency stack in a vector of strings*
12. **v=stack():** *return the dependency stack in a vector of strings*

As a string

Return the dependency name

As an integer or a float

Return the dependency id.

Example

```
//XIP rule
|Noun#1| if (subj$1(#2,#1) & testdep($1))…

//KiF code
Function testdep(dependency d) {
        print("D=",D,"\n");          //display: SUBJ
        map features;
```

```
                d.features(features);
                vector v;
                v=d.parameters();
                return(true);
        }
```

Type generation

This object shares most of its methods with *dependency. However, it adds the following one:*

Methods

1. **child():** *return the first Generation Node child*
2. **last():** *return the last child*
3. **next():** *return the next Generation Node*
4. **parent():** *return the parent Generation Node*
5. **previous():** *return the previous Generation Node*
6. **rulenumber():** *return a vector of the rule numbers, which applied to create this generation node.*

As a string

Return the generation name

As an integer or a float

Return the generation id.

Example

```
//XIP rule
|Noun#1| if (NP$#1(#2,#1) & testdep($#1))…

//KiF code
Function testdep(generation d) {
        print("D=",D,"\n");          //display: NP
        map features;
        d.features(features);
        return(true);
}
```

Type: graph

This object is used to handle XIP conceptual graphs, it exposes many methods to project, extract or modify conceptual graphs. The creation and initialisation of the graph must be done in the XIP grammar.

Methods

1. **match(extractor):** extractor *should be a graph. We return a vector of all sub-graphs from the current graph that match* extractor*.*
2. **name():** *return the name of the graph*
3. **pop(remove):** remove *should be a graph. We remove from the current graph, the sub-graphs that match remove.*
4. **project(g):** g *should be a graph. We project* g *on the current graph.*
5. **replace(pattern,replacement):** pattern *and* replacement *should be two graphs. We replace the sub-graph that matches pattern with a new replacement graph.*
6. **set(string name):** *give the graph a name*

Operators

Graphs can be compared with the operators ‘==’ and ‘!=’.

As a string

We return the name of the graph.

Type xiptrans

The “xiptrans” type is a finite-state transducer implementation, which enables the implementation of large lexicons compatible with XIP. It supplies some methods to create and test these lexicons before integrating them into a XIP grammar.

Methods

1. **build(string input,string output):** *Build a xiptrans file out of a text file containing on the first line surface form, then on next line lemma+features.*

2. **compilergx(string rgx,svector features):** *Build a xiptrans file out of regular expressions. See below for a description of these very basic expressions.*

3. **compilergx(string rgx,svector features,string name):** *Build a xiptrans file out of regular expressions. See below for a description of these very basic expressions. It Stores the automaton in a filename.*

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

5. **lookup(string wrd):** *Lookup of a token using a xiptrans automaton.*

6. **lookup(string wrd,int threshold,int flags):** *Lookup of a token using a xiptrans automaton, with a threshold and flags.*

    - **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.*

    - **a_change:** *the automaton can change a character to another*

    - **a_delete:** *the automaton can delete a character*

    - **a_insert:** *the automaton can insert a character*

    - **a_switch:** *the automaton switches two characters*

    - **a_nocase:** *the automaton takes into account the difference in case with the current character string.*

7. **parse(string sentence):** *Lookup of a token using a xiptrans automaton.*

Building

A file should be organized in the following way:

a) The first line is the surface form of your word. This line may contain spaces, which will be part of the string itself.

b) The second line is the lemma, a tab and a list of features, each starting with a “+”. The lemma should not contain any spaced.

Examples:

```
best
best            +int_adj
best
best            +Verb+Trans+Pres+Non3sg+VERB
best
best            +Noun+Sg+NOUN
best
good            +Adj+Sup+ADJSUP
best
well            +Adv+Sup+ADVSUP
best known
best_known      +Adj+ADJ
best of breed
best_of_breed   +Noun+Sg+NOUN
best-known
best-known      +Adj+ADJ
```

Regular expressions

xiptrans compiles a regular expression, with on the one hand a regular expression and on the other hand a vector of features.

A regular expression accepts the following structure:

- $ccc is a string
- [c c c]: is a disjunction of characters
- [c c c]+ is a disjunction of characters repeated at least once
- [x-y] is a range of values between x and y
- [x-y]+ is a range of values between x and y at least once
- {s s s} is a disjunction of strings
- {s s s}+ is a disjunction of strings repeated at least once
- %c: is one character
- (..) is an optional list of structures.

- !d is a reference to one of the features in the vector of features.

Example:

```
//We declare a xiptrans object
xiptrans tr;

//Our regular expression to recognize numbers.
string rgx="([- +])[0-9]+!1(%.[0-9]+!2";
//two features, +Num matches !1 and +Dec matches !2 in the above expression
svector vfeats=["+Num","+Dec"];
tr.compilergx(rgx,vfeats);

svector vs=tr.lookup("1.234"); //yields 1.234        +Dec
vs=tr.lookup("1234"); //yields 1234          +Num
```

Type fst: Xerox Finite-state transducers

This type is very specific to XIP and is not available with all version of the rule engine, depending on its license. With this type, it is possible to load a finite-state transducer script and to apply it to any string. It is also possible to create an FST on the fly.

Methods

1. **add(string surface,string lemma):** *Store a new* surface/lemma *couple in the FST, which has been prepared with* init()*.*
2. **compact(int option):** *Compact the current net: 1 is SMALLEST, 2 is SMALL, 3 is FAST, 4 is FASTEST.*
3. **compile(map m, boolean utf8):** *compile the map m into a FST. Keys should be a lemma+"\t"+features and values should be a surface form:* m["abc\t+Masc+Noun"]="abc". COMPILE *is basically a call to init(), then an iteration on the map with add(...) with a final save.*
4. **compile(vector v, boolean utf8):** *compile the vector v into a FST. Values should alternate, first lemma+"\t"+features and then surface forms:* ["abc\t+Masc+Noun","abc",..]. COMPILE *is basically a call to init(), then an iteration on the vector two values at a time with add(...) with a final save.*
5. **compilescript(string file,bool utf8):** *Compile a file containing regular expressions.*
6. **compilewith(map m, string filename,boolean utf8):** *Store the map as an FST spaced-text format file. It also compiles the map into an FST in memory. The map should follow the same rule as above.*
7. **down(string l, string feats):** *return the surface form of a lemma l with features feats*
8. **FST var(string script):** *create and load an FST script. Strateg is DEPTH strategy, flags is "". UTF8 is false.*
9. **FST var(string script, boolean utf8):** *create and load an FST script. Strategy is DEPTH strategy, flags is "".*
10. **FST var(string script, string flags, int strategy,boolean utf8):** *create and load an FST script. Strategy=0 for a DEPTH strategy, 1 for a BREADTH strategy.*

11. **init(boolean utf8):** *Initialization of a net creation, step by step. If the net should contain UTF8 characters, set utf8 to* true*.*

12. **invert():** *Invert the current net.*

13. **load(string script, string flags, int strategy,boolean utf8):** *load an FST script. Strategy=0 for a DEPTH strategy, 1 for a BREADTH strategy.*

14. **loadfst(string fstfile, string flags, int strategy,boolean utf8):** *load an FST file. Strategy=0 for a DEPTH strategy, 1 for a BREADTH strategy.***loadregex(string file,bool utf8):** *Load a file containing a regular expression.*

15. **loadspacedtext(string filename,bool utf8):** *load a text file in which words are stored according to the FST spaced-text format.*

16. **minimize():** *minimize the arcs in the current net*

17. **negate():** *return the negated automaton out of the current automaton*

18. **optimize():** *Optimize the arcs in the current net*

19. **save(string filename,boolean utf8):** *final operation after an* init() *and a series of* add(). *Store the final FST in* filename.

20. **traverse(string pathname, function f,imap grds,boolean lower)***: traverse a FST with* f(string s,imap grds) *called for each new token found in the FST of name pathname, on the FST lower side if* lower *is true.*

21. **uncompact():** *Uncompact the current net.*

22. **unoptimize():** *Unoptimize the arcs in the current net*

23. **unvectorize():** *Unvectorize the current net.*

24. **up(string w):** *return the vector of all readings for the word w*

25. **up(string w,int threshold,int flags):** *return the vector of all readings for the word w, with a threshold and some edit distance flags, to detect words which are close to w to an edit distance set of actions. The flags can have the following values:*

    a. **a_change:** *modification of characters*

    b. **a_delete:** *deletion of characters*

c. **a_insert:** *insertion of characters*

d. **a_last:** *compute the edit distance only for the last FST of a cascade*

e. **a_prefix:** *check if a string is a prefix in the FST*

f. **a_switch:** *switching two characters next to each other*

26. **vectorize():** *Vectorize the current net.*

As a string

Return the filename of the FST script

Operator mapping

The following operators have been mapped to specific automaton operations:

a1 | a2: *return the union of two automata*

a1 & a2: *return the intersection of two automata*

a1 + a2: *return the concatenation of two automata*

a1 - a2: *return the difference between two automata*

a1 * a2: *return the composition of two automata*

The result is always a *fst* object.

Example

```
words["toto\t+Noun"]="titi";
words["titi\t+Noun"]="tutu";

myfst.compile(words);
res=myfst.up("titi");
print("Res=",res,"\n");      //display: ['toto    +Noun']


//Another possibility is to create the FST step by step...
fst myfstbis;
//First we prepare our FST
myfstbis.init();
iterator it=words;
//Then we add line after line: surface,lemma
for (it.begin();it.nend();it.next())
        myfstbis.add(it.value(),it.key());
//Eventually we save it
myfstbis.save("myfstbis.fst");
res=myfstbis.up("tutu");
print("Res=",res,"\n");      //display: ['titi      +Noun']
```

Type ntm

ntm is a specific library based on the FST, which articulates in one single step, morphological analyzis, normalization and tokenization.

Methods

ntm exposes the following methods:

1) **loadntm(string script,bool utf8):** *load a NTM script. The second parameter is optional*

2) **lookupstring(string sentence)**: *We apply the ntm script onto a string. The result is a vector of vectors.*

3) **lookupfile(string filename):** *We apply the ntm script onto a file. The result is a vector of vectors.*

4) **setsepconstraint(bool v):** *We set the separator constraint in NTM.*

5) **setoptions(map opt):** *We set the following options:*

   a. **“utf8”:** *UTF8 mode*

   b. **“nsc”:** *Boolean (non separator constraint)*

   c. **“vect”:** *Integer (vectorization threshold)*

   d. **“unknownbychars”:** *Boolean (false=BYTOKEN, true=BYCHAR)*

Loading

You load an ntm script at creation time:

```
ntm n('C:\XIP\Test\kifcxx\phonologie\ntmscriptkif',true);
```

The first parameter is the ntm script pathname. The second parameter is the lexicon encoding.

a) true: the encoding is utf8

b) false: the encoding is not utf8

Example

```
ntm n('C:\XIP\Test\kifcxx\phonologie\ntmscriptkif',false);

string s="The lady is happy.";
```

```
v=n.lookupstring(s);
println(v);
```

Result:

```
[['The   the      +Det+Def+SP+DET'],
['lady   lady     +Noun+countable+c_person+Sg+NOUN'],
['is     be       +Verb+Pres+3sg+VBPRES'],
['happy  happy    +Adj+s_sc_pwith+s_sc_pabout+ADJ'],
['.      .        +Punct+Sent+SENT']]
```

Type parser

KiF provides a specific type to load a XIP grammar from a KiF program. It is then possible to parse a string with a specific grammar and display the result of the analysis. In the case of a call of a KiF program from an application using XIP as an external library, the results can be hacked through a callback method.

kif_exchange_data is any object that is passed to XIP, which will then be available within the grammar as the XIP variable: *kif_exchange_data*. Thanks to this variable, it is possible to pass an object through a grammar and use it in a KiF function that would be called from within this grammar.

Methods

1. **addendum(string filename):** *Load an addendum file into the grammar to enrich it.*

2. **addendum(string rules,bool add):** *Compile a set of XIP rules and add them at the end of the current grammar, or merge them into the given layers.*

3. **addoption(int nb1,int nb2,...):** *Add the options of id nb1,nb2 ... See below for a list of these options.*

4. **distance():** *Return a vector of distances between nodes, for the current analysis*

5. **distances():** *Return a vector of distances between nodes, for all analyses***.**

    - *Requires the flag: XIP_COMPUTE_DISTANCE.*

6. **generatefromfile(string input,kif_exchange_data,string output):** *generate a sentence out of a dependency file*

7. **generatefromstring(string input,kif_exchange_data):** *generate a sentence out of a group of dependencies*

8. **generaterule(string depname,vector focus,vector nodes,vector feats,vector dependencies,vector dependencyfeatures,bool addquestionmark,int typerule):** *generate a dependency rule out of a vector of* **node,** *their associated features, a vector of dependencies and their features. If two nodes are separated by a sequence of nodes, then with addquestionmark, this sequence is replaced with "?+" in the rule, or by the category sequence itself otherwise. (see below for an*

example). addquestionmark *and* typerule *are optional. Their default values are respectively* **false** *and* **0***.*

9. **getgrammar(int handler):** *you can initialize a parser with a grammar that has been loaded outside the KiF program. If you do not provide the handler, then the grammar is the current one.*

10. **getrulebody(int n):** *return the rule body according to the rule number (see rulenumber for node and dependency).*

11. **grammarfiles():** *return a map of all files in a grammar*

12. **load(string grmpathname):** *Load a XIP grammar.*

13. **lookup(string s):** *Apply a simple lookup on the string (similar to ntmonly in XIP). To disable the tagging process, you can use* **removeoption(XIP_TAGGER).**

14. **name():** *Return the pathname of the grm file*

15. **upto(int nbrule):** *run the grammar up to a certain rule number*

16. **parse(string sentence,kif_exchange_data):** *parse a sentence using the grammar. If no variable is provided to get the analysis, then the result is displayed on the current output.*

17. **parsefeatures(svector finit,smap fres):** *translate the FST features in finit into XIP features, to store them in fres. Return the Part of Speech of the expression if found as a string.*

18. **parsefile(string input,kif_exchange_data,string output):** *parse the file* input *and store the results in string* output.

19. **parsefile(string input,kif_exchange_data,file output):** *parse the file* input *and store the results in file* output. Output *must have been created as a write file:* file ouput(fname,'w');

20. **parser var(string grmpathname):** *Create a parser object and Load a XIP grammar.*

21. **parsestructure(map m|vector v):** *parse a predefined tree using the grammar. The tree can also be a vector. The structure should respect the output of map or of vector on a 'node'.*

22. **parsexml(string input,kif_exchange_data, int depth):** *parse the XML file* input *A callback function must be provided in that case to catch the XipResult objects.*

23. **parsexml(string input,kif_exchange_data,string output,int depth):** *parse the XML file* input *and store the results in file* output. Depth *is the depth at which the analysis should take place.*

24. **parsexmlstring(string input,kif_exchange_data, int depth):** *parse the XML string* input*.*

25. **reload():** *reload the grammar.*

26. **removeoption(int nb1,int nb2,...):** *Remove the options of id nb1,nb2, nb3 ...*

27. **setcatchxml(function):** *set the catch xml function, which is used to catch the extracted string from the XML XIP internal module. This function should return the new string that will be analyzed. The function should have the following signature:*

    - **function mycatch(string text) {.... return(newtext);}**

28. **texttoxml(string input,kif_exchange_data,string tag,string encoding):** *transform a text file into an XML file, using 'tag' as the new xml root and 'encoding', which can take either 'latin' or 'utf8' as a value*

Options

XIP uses the following options to guide the parsing process. Each of the following option is a value, which can be used in your initialization process.

XIP_ENABLE_DEPENDENCY
XIP_LEMMA
XIP_SURFACE
XIP_MARKUP
XIP_TAGGER
XIP_ENTREE
XIP_CATEGORY
XIP_REDUCED
XIP_FULL
XIP_OFFSET
XIP_WORDNUM
XIP_SENTENCE
XIP_NONE
XIP_DEPENDENCY_BY_NAME
XIP_DEPENDENCY_BY_NODE
XIP_DEPENDENCY_BY_CREATION

XIP_TREE
XIP_TREE_PHRASE
XIP_TREE_COLUMN
XIP_MERGE_XML_SUBTREE
XIP_CONVERSION_UTF8
XIP_EXECUTION_ERROR
XIP_MATHEMATICS
XIP_DEPENDENCY_NUMBER
XIP_UTF8_INPUT
XIP_EXECUTE_TOKEN
XIP_SENTENCE_NUMBER
XIP_LANGUAGE_GUESSER
XIP_NOT_USED
XIP_CHUNK_TREE
XIP_DEPENDENCY_FEATURE_VALUE
XIP_NO_TAG_NORMALISATION
XIP_LOWER_INPUT
XIP_CHECK_INPUT_UTF8
XIP_GENERATION_CATEGORY
XIP_GENERATION
XIP_RANDOM_ANALYSIS
XIP_ JSON

Executing External Functions

It is also possible to call an external KiF function through a *parser* object. If a grammar is loaded, which implements a KiF program, then the global functions in that KiF program can be executed from our local KiF program. Beware that *function checking* will be done at run time.

Example

```
parser p;
p.load('c:\grammar\english.grm'); //we load a grammar implementing Read
string s=p.Read("xxx"); //we can execute Read in our local program.
```

Generating rules

KiF exposes the method “generaterule” to generate dependencies rules based on some nodes from the chunk tree, and a list of dependencies extracted so far.

IMPORTANT: *This method can only be called in a function executed from within XIP, when nodes and dependencies are still alive.*

generaterule(string depname,vector focus,vector nodes,vector feats,vector dependencies,vector dependencyfeatures,bool addquestionmark,int typerule)

Parameters:

1) depname *is the name of the dependency that will be generated as output of the rule.*

2) focus *is a vector of XIP nodes, which are used in building the dependencies arguments.*

3) nodes *is the vector of XIP nodes (see type node), which are supplied to the method, to detect which nodes will be used to build our rule.*

4) feats *is a vector of sub-vectors, where each sub-vector is associated with its corresponding nodes in the vector of XIP nodes. Each sub-vector should contain a list of strings of the form: "attribute:value". Each can be empty.*

5) dependencies *is a vector of dependency objects (see type dependency). They will be integrated into a "if" section in the rule when they are provided.*

6) dependencyfeatures *is a vector of sub-vectors (as* feats), *where each sub-vector corresponds to a dependency above. Each sub-vector should contain a list of strings of the form: "attribute:value". Each can be empty.*

7) addquestionmark *is a Boolean, which defines whether the system should replace a sequence of categories with a "?+".*

8) typerule is an integer, which defines the rule type that will be created.

```
a. DEPENDENCYRULE       0 (default value)
b. SEQUENCERULE         1
c. LEFTCONTEXT          2
d. RIGHTCONTEXT         3
e. IDRULE               4
f. TAGGINGRULE          5
```

Example:

```
function createrule(node n1,node n2) {
    //First we extract the features
    mapss f1=n1.features();
    mapss f2=n2.features();

    //We then create our feature list. We take ALL features to create our rule
    string f;
    svector v1;
    for (f in f1)
        v1.push(f+":"+f1[f]);
    svector v2;
    for (f in f2)
        v2.push(f+":"+f1[f]);

    //We need a vector of sub-vectors
    vector myfeatures=[v1,v2];
    vector mynodes=[n1,n2]; //our nodes

    //Our dependency output will be TOTO, we do not provide dependencies
    string rule=myparser.generaterule("TOTO",mynodes,myfeatures,[],[],true,0);
    println(rule);
}
```

```
//Our XIP rule to call createrule
string rule=@"

script:

   //Our nodes are connected through a SUBJ dependency
   if (subj(#1,#2)) {
      createrule(#2,#1);
   }
"@;

myparser.addendum(rule,true);
```

As a string

Return the pathname of the *grm* file

As an integer

Return the integer handle of the grammar

As a Boolean

Return *true* if a grammar has been loaded.

Example:

```
parser p;
p.load('/home/user/grammar.grm');
string s=p.parser("This is an example.",null);
print("S=",s,"\n");
```