Xerox Incremental Parser

KiF Language

xerox

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 which is part of the XIP engine.

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 are introduced anywhere with a //.

```
//This is a comment
```

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.

Anticipated declarations

Functions and frames can be declared in advance, before their actual implementations.

Examples

```
function toto(int i);
frame test;
```

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.

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);
```

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 two specific systems variables: *_args* and *_paths.*

Example:

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

System vector: _args

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: _paths

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.

Separator in pathnames: _sep

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.

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.

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:

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

```
//Kif section
Kif:

//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.

autorun

An *autorun* function is automatically launched after its declaration.

If you declare an *autorun* function in a frame, then you must declare an *_initial()* function in that frame, even though such a function does not seem necessary to your code. Furthermore, *autorun* functions will be executed *before* any specific code in your *_initial* function. (*see WAITALL* for an example of such a use within a frame).

thread

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

Example:

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

//A function is called by stating its name with its parameters
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);
```

join and waitonjoin

A thread can be declared as *join*, 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 w*aitonjoin()* which will then wait for these threads to finished.

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

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. For instance, 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 prevalent.

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

polynomial 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 example

```
autorun waitonred() {
	while (true) {
		wait("red");
		println("What nice colour!!!");
	}
}
```

This function will be automatically launched, waiting on the string: “red”.

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 written 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 | autorun | 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 structure 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++
```

Initialization Order

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

Furthermore, if a frame field 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 on 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 codes
//All we need is to use the same frame instruction as above, adding some new stuff

frame myframe  {
```

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

Functions pre-declaration

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

```
//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 others.

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

```
//essai d'appel de 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
```

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. Some of these objects are specific to XIP. 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*

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.

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 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
?	Stands for any character
% ?	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

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.

Methods

1. **size():** *return the length of a string*
2. **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.*
3. **bytes():** *return a vector of bytes matching the string.*
4. **split(string splitter,vector vect):** *split a string along splitter and store the results in a vector. If splitter==“”, then the string is split into a vector of characters*
5. **removefirst(int nb):** *remove the first nb characters of a string*
6. **removelast(int nb):** *remove the last nb characters of a string*
7. **left(int nb):** *return the first nb characters of a string*
8. **right(int nb):** *return the last nb characters of a string*
9. **mid(int pos,int nb):** *return the nb characters starting at position pos of a string*
10. **extract(int pos,string from, string up1,string up2...):** *return a vector 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.*
11. **reverse():** *reverse the string*
12. **fill(int nb,string char):** *create a string of* nb chars.
13. **byteposition(int pos):** *convert a character position into a byte position (especially useful in UTF8 strings)*
14. **charposition(int pos):** *convert a byte position into a character position (especially useful in UTF8 strings)*
15. **pop(i):** *remove character at position i*

16. **pop():** *remove last character*
17. **last():** *return last character*
18. **insert(i,s):** *insert the string s at I*
19. **trim():** *remove the trailing characters*
20. **trimright():** *remove the trailing characters on the right*
21. **trimleft():** *remove the trailing characters on the left*
22. **upper():** *Return the string in upper characters*
23. **lower():** *Return the string in lower characters*
24. **islower():** *Test if a string only contains lowercase characters*
25. **isupper():***Test if a string only contains uppercase characters*
26. **isalpha():***Test if a string only contains only alphabetical characters*
27. **isdigit():** *Test if a string only contains digits*
28. **isutf8():** *Test if a string contains utf8 characters*
29. **latin():** *convert an UTF8 string in LATIN*
30. **utf8():** *convert a LATIN string into UTF8*
31. **find(string sub,int pos):** *Return the position of substring sub starting at position pos*
32. **rfind(string sub,int pos***): Return the position of substring sub backward starting at position pos*
33. **replace(sub,str):** *Replace the substrings matching sub with str*
34. **replacergx(rgx,str):** *Replace the substrings matching rgx with str*
35. **splitrgx(rgx***): Split string with regular expression rgx. Return a vector of substrings. Need regular expression operator (…) to keep substrings.*
36. **boolean=regex(rgx***): Test if the regular expression rgx applies to string*
37. **vector=regex(rgx):** *Return all substrings matching rgx*

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

39. **int=regex(rgx):** *Return the position of the substring matching rgx in the string*

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

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

41. **boolean=regexip(rgx):** *Test if the regular expression rgx applies to string*

42. **vector=regexip(rgx):** *Return all substrings matching rgx*

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

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

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

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

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

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. Note that if the last substring is not found, then the string will be extracted up to the end.*

As an integer or a float

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

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=1234
x=s["2":"a"];           //value=2345678
s[2]="ef";              //value=12EF45678a

//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"]
//Equivalent and a little bit faster
s.split(" ",v);         //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']
```

Type: int, float, long

KiF provides three different numerical: *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.

Methods:

1. **log():** *return the log base 10 of the value*
2. **ln():** *return the neperian log*
3. **exp():** *return the exponential*
4. **sqrt():** *return the square root*
5. **tan():** *tangent*
6. **sin():** *sinus*
7. **cos():** *cosinus*
8. **atan():** *arc tangent*
9. **chr():** *return the ascii character corresponding to this number as a code.*
10. **#():** *return the bit complement*
11. **format(string form):** *return a string formatted according to the pattern in form. (this format is the same as the sprintf format in C++)*
12. **get():** *Read a number from keyboard*
13. **fraction():** *return the value as a fraction.*

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 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

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.*

#(value): *returns the complement of the bit 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

#(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. **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.*
2. **n():** *return the numerator of the fraction*
3. **d():** *return the denominator of the fraction*
4. **n(int v):** *set the numerator of the fraction*
5. **d(int v):** *set the denominator of the fraction*
6. **nd(int n,int d):** *set the numerator and denominator of a fraction*
7. **invert():** *switch the denominator with the numerator 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. **size():** *return the length of the vector*
2. **test(int i):** *test if i is a valid slot in the vector*
3. **push(a):** *add* a *to the vector*
4. **pop(int i):** *remove the ith element from the vector and return it.*
5. **pop():** *remove the last element from the vector and return it.*
6. **last():** *return the last element of the vector*
7. **reverse():** *reverse the order of the elements in the vector*
8. **join(string sep):** *concatenate each element in the vector in a string where each element is separated from the others with sep*
9. **bytes():** *return the string matching the bytes stored in the vector. The vector should only contains integers between 0..255.*
10. **insert(i,x):** *insert the element x at position i*
11. **sum():** *Sum each element with the others*
12. **product():** *Multiply each element with the others*
13. **clear():** *clean the vector*
14. **sort(compare):** *sort the content of the vector according to* **compare** *function.*
15. **sortint(bool order):** *sort the content of the vector, forcing each value to be a int. order==true order is increasing, order=false order is decreasing.*
16. **sortfloat(bool order):** *sort the content of the vector, forcing each value to be a float. order==true order is increasing, order=false order is decreasing.*

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

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

19. **apply(a,b,c…):** *apply all functions or XIP rules stored in the vector, passing a,b,c etc. as parameters.*

20. **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.

21. **range(first, last):** *generate a vector of integer from* first *to* last, *with an step of 1.*

22. **range(first, last,step):** *generate a vector of integer from* first *to* last, *with an step of* step.

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']
```

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*

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.

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
```

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. **size():** *return the length of the list*
2. **test(int i):** *test if i is a valid slot in the list*
3. **pushfirst(a):** *add* a *to the beginning of the list*
4. **pushlast(a):** *add* a *to the end of the list*
5. **popfirst():** *remove the first element from the list and return it.*
6. **poplast():** *remove the last element from the list and return it.*
7. **first():** *return the first element of the list*
8. **last():** *return the last element of the list*
9. **reverse():** *reverse the order of the elements in the list*
10. **join(string sep):** *concatenate each element in the list in a string where each element is separated from the others with sep*
11. **insert(i,x):** *insert the element x at position i*
12. **clear():** *clean the list*
13. **apply(a,b,c…):** *apply all XIP rules stored in the list, passing a,b,c etc. as parameters.*
14. **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.
15. **sum():** *Sum each element with the others*
16. **product():** *Multiply each element with the others*

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 map

The 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.

Methods

1. **size():** *return the length of the map*
2. **test(string k):** *test is k is a valid key in the map*
3. **sum():** *Sum each element with the others*
4. **product():** *Multiply each element with the others*
5. **pop(string key):** *remove the elements matching key*
6. **keys():** *returns the map keys as a vector*
7. **values():** *return the values as a vector*
8. **clear():** *clean the vector*
9. **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.
10. **apply(a,b,c…):** *apply all XIP rules stored in the map, passing a,b,c etc. as parameters.*
11. **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.

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 second way is to test whether the return value for a given key is *null.* However, in the last case, the KiF philosophy might blur the understanding of that test a little bit.

First of all, one should remember that ""==null or 0==null returns always true, since "" is the string value of null and 0 is the integer value of null.

Thus, in the following example, it will be impossible to differentiate the two tests:

```
map m={"a":1,"n":2,"u":""};
```

```
if (m["u"]==null)
      println('u is a key whose value is ""');

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

The only way to force the right test is to force a test on *null* and not on its string or integer interpretation. We simply invert the test. In that case, there is no possible interpretation as a string or as an integer of the right side of the equality. The test will only succeed in m["ee"] returns *null*.

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

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 ");
```

Type tree

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

Methods

1. **tree(value):** *create a tree node, with* value *as a value.* Value *can have any types*
2. **tree n=100:** *modify the value of a tree node with anything, here with an integer, but it could any object.*
3. **nextnode():** *return the next tree node as a* tree *object*
4. **previousnode ():** *return the first tree node as a* tree *object*
5. **childnode ():** *return the first tree node as a* tree *object*
6. **parentnode ():** *return the parent tree node as a* tree *object*
7. **lastnode ():** *return the last child tree node as a* tree *object*
8. **nextnode (tree n):** *compare the next tree node with* n
9. **previousnode (tree n):** *compare the previous tree node with* n
10. **childnode (tree n):** *Test if the current node is a child of* n
11. **parentnode (tree n):** *Test if the current node is a parent of* n
12. **lastnode (tree n):** *Test if the current node is the last child of* n
13. **addchildnode (tree n):** *Add n at the end of the children of the current tree node*
14. **addnextnode (tree n):** *Add n after the current tree node*
15. **addpreviousnode (tree n):** *Add n before the current tree node*
16. **isolatenode ():** *Extract the current node from its tree*
17. **removenode ():** *Delete the current sub-tree from the global tree*
18. **depth():** *Return the depth of the node in the tree*

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.childnode()!=null) {
                print("("); //subnodes are displayed between "(…)"
                treedisplay(t.childnode());
                print(")");

        }
        treedisplay(t.nextnode());
}

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

test1.addchildnode(test2);
test1. addchildnode (test3);
test2. addchildnode (test4);
test4.addnextnode(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.removenode();
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 matrix

This type is very specific. It is used to store values in a matrix where each line is a specific instance with a given class and a given attribute.

Methods

1. **size():** *return the number of elements in the matrix*
2. **instances():** *return the number of instances in the matrix*
3. **set(int instance,string class,string attribute, value):** *Add an element with a class instance value, according to an attribute and an instance Id.*
4. **get(int instance,string class,string attribute):** *Return the element with a specific class instance value, according to an attribute and an instance Id.*
5. **clear():** *clean the vector*
6. **save(filename,type):** *save the matrix on the disk.* **type** can take two values: “arff” or “xml”.

Operator

x in amatrix: *return true if x is in amatrix values*

As an integer or a float

Return the number of elements stored in the matrix

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

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. **setdate(year,month,day,hour,min,sec):** *set a time variable*
2. **date():** *return the date as a string*
3. **year():** *return the year as an integer*
4. **month():** *return the month as an integer*
5. **day():** *return the day as an integer*
6. **hour():** *return the hour as an integer*
7. **min():** *return the min as an integer*
8. **sec():** *return the sec as an integer*

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

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

This type is used to manage a file in input and output.

Methods

1. **file f(string filename, string moderead):** *open a file according to* moderead*. The possible values for* moderead *are:*
    a. **“r”:** read
    b. **“w”:** write
    c. **“a”:** append
    d. **“w+”:** append
2. **openwrite(string filename):** *open a file in write mode*
3. **openappend(string filename):** *open a file in append mode*
4. **openread(string filename):** *open a file in read mode*
5. **write(string s1,string s2,…):** *write strings in the file*
6. **writebin(int s1,int s2,…):** *write bytes in the file*
7. **get():** *read one character from the file*
8. **unget():** *return one character to the stream*
9. **unget(nb):** *return nb character to the stream*
10. **read():** *read a line from a file*
11. **readln():** *read a line from a file*
12. **seek(int p):** *position the file cursor at p*
13. **tell():** *return* the position of the file cursor
14. **eof():** *return* true *when the end of file is reached*

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;
vector words;
string w;
for (s in f) {//Using the in operator
        s=s.trim();
        s.split(" ",words);
        for (w in words)
                print("word:",w,endl);
}
f.close();
```

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 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. **name():** *return the part of speech*
2. **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).*
3. **data(map feats):** *Features are stored in map as attribute/value*
4. **setfeature(string att, string val):** *feature assignment to the node*
5. **feature(string att):** *return the value of an attribute*
6. **feature(string att,string val):** *test the value of an attribute*
7. **next():** *return the next node after current node*
8. **previous():** *return the previous node after current node*
9. **child():** *return the first child node under current node*
10. **last():** *return the last child node under current node*
11. **parent():** *return the parent node above current node*
12. **surface():** *return the surface string*
13. **surface(string s):** *replace the surface string of the current node with s.*
14. **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.*
15. **lemma(string s):** *replace the lemma of the current node with s.*
16. **number():** *return the ID of current node*
17. **offset(int left,int right):** *left and right receive the offset*
18. **offsetchar(int left,int right):** *left and right receive the character offset*

19. **tokenoffset(int left,int right):** *left and right receive the token offset*
20. **leftoffset():** *return the left offset*
21. **rightoffset():** *return the right offset*
22. **leftoffsetchar():** *return the left character offset*
23. **rightoffsetchar():** *return the right character offset*
24. **righttokenoffset():** *return the right token offset*
25. **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.data(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. **name():** *return the dependency name*
2. **data(map feats):** *Features are stored in map as attribute/value*
3. **setfeature(string att, string val):** *feature assignment to the dependency*
4. **feature(string att):** *return the value of an attribute*
5. **feature(string att,string val):** *test the value of an attribute*
6. **parameters():** *return a vector of* node *variable*
7. **stack(vector v):** *return the dependency stack in a vector of strings*
8. **push(string s):** *push s on the dependency's stack*
9. **pop(int i):** *remove the i^{th} element from the dependency's stack*
10. **pop():** *remove the last element on top of the dependency's stack*

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.data(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. **next():** *return the next Generation Node*
2. **previous():** *return the previous Generation Node*
3. **child():** *return the first Generation Node child*
4. **parent():** *return the parent Generation Node*
5. **last():** *return the last child*

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.data(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. **set(string name):** *give the graph a name*
2. **name():** *return the name of the graph*
3. **project(g):** g *should be a graph. We project* g *on the current graph.*
4. **pop(remove):** remove *should be a graph. We remove from the current graph, the sub-graphs that match remove.*
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. **match(extractor):** extractor *should be a graph. We return a vector of all sub-graphs from the current graph that match* extractor.

Operators

Graphs can be compared with the operators '==' and '!='.

As a string

We return the name of the graph.

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. **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.*

2. **load(string script, string flags, int strategy,boolean utf8):** *load an FST script. Strategy=0 for a DEPTH strategy, 1 for a BREADTH strategy.*

3. **init(boolean utf8):** *Initialization of a net creation, step by step. If the net should contain UTF8 characters, set utf8 to* true.

4. **add(string surface,string lemma):** *Store a new* surface/lemma *couple in the FST, which has been prepared with* init().

5. **save(string filename,boolean utf8):** *final operation after an* init() *and a series of* add(). *Store the final FST in* filename.

6. **compile(map m, string filename,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.*

7. **up(string w):** *return the vector of all readings for the word w*

8. **down(string l, string feats):** *return the surface form of a lemma l with features feats*

As a string

Return the filename of the FST script

Example

```
words["toto\t+Noun"]="titi";
words["titi\t+Noun"]="tutu";

myfst.compile(words,"myfile.fst");
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 xml

This type is used to handle XML nodes, which are provided through a XIP XML expression. This type matches the *xml* type of XIP. Furthermore, XIP XML expressions are also part of the KiF language.

Methods

1. **name():** *return the XML node name*
2. **data(map props):** *Properties are stored in map as attribute/value*
3. **next():** *return the next XML node*
4. **previous():** *return the previous XML node*
5. **child():** *return the first child node under current node*
6. **parent():** *return the parent node above current node*

As a string

Return the XML node name

Example

```
//KiF language
xml xnode;
@data(/root/val))->Set(xnode);
```

The @data must have been declared in XIP as an alias to an XML file.

Type rule

It is also possible to execute a XIP rule from a KiF script. These rules can be either defined in a KiF section or in XIP file.

In a XIP file, use the keyword: *KifDependencies*, *with as a next instruction:* rule myrules, *where* myrules *is a kif variable.*

Only dependency rule can be stored in this way.

Each next rule will then be stored in the inner vector of that rule variable. This variable can be declared as many times as necessary to store more rules in it. More than one variable can be declared for a given grammar.

Methods

1. **apply(a,b,c…):** *apply a rule or a group of rules*, passing parameters to any KiF functions that might be called back from the rule itself.
2. **set(float weight,float threshold,float value):** *set the probabilities value of a rule*
3. **get(float weight,float threshold,float value):** *get the probabilities value of a rule*

Example 0

```
KifDependency:
rule myrules;
```

All the rules declared within this section after this declaration will be stored in *myrules*.

```
if (….)…
```

Example 1

Rules can be directly declared in a KIF programme in the following way:

```
rule myrule = |Noun#1,Verb#2| if (~subj(#2,#1)) subj(#2,#1);
```

To apply a rule, you simply call the *apply* method: *myrule.apply();*

Example 2

It is also possible to declare a rule, which in turn will call a KIF function.

```
rule myrule = |Noun#1,Verb#2|
if (~subj(#2,#1)) {kifcall(x,y);};
```

x,*y* parameters can then be passed through the apply function: *myrule.apply(10,20)*, from which *kifcall* can benefit.

Example 3

Rules can also be applied one after the others with an *iterator*:

```
iterator it=myrule;
for (it.begin();it.nend();it.next())
  it.apply(10,20);
```

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. **parser var(string grmpathname):** *Create a parser object and Load a XIP grammar.*
2. **load(string grmpathname):** *Load a XIP grammar.*
3. **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.*
4. **parsefile(string input,kif_exchange_data,string output):** *parse the file* input *and store the results in file* output.
5. **parsefile(string input,kif_exchange_data):** *parse the file* input *within a XIP library call. A callback function must be provided in that case to catch the XipResult objects.*
6. **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.*
7. **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.*
8. **generatefromstring(string input,kif_exchange_data):** *generate a sentence out of a group of dependencies*
9. **generatefromfile(string input,kif_exchange_data,string output):** *generate a sentence out of a dependency file*

10. **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*

11. **name():** *Return the pathname of the grm file*

12. **addoption(int nb1,int nb2,...):** *Add the options of id nb1,nb2 ... See below for a list of these options.*

13. **removeoption(int nb1,int nb2,...):** *Remove the options of id nb1,nb2, nb3 ...*

14. **addendum(string filename):** *Load an addendum file into the grammar to enrich it.*

15. **grammarfiles():** *return a map of all files in a grammar*

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_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

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.
```

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");
```

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. **load(string kifpathname):** *Load a KiF program*
3. **name():** *Return the pathname of the grm file*
4. **exposed():** *Return a vector of exported function by the KiF program. Each element is a function.*
5. **_loader:** *A* kif *variable (of the type described here), which stores a pointer to the loading program.*
6. **open():** *this function opens a session and return a session handler.*
7. **clean():** *this function closes a session and clean it.*
8. **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.*
9. **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.*

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.

Example

```
loadin('c:\files\prgm.kif'); //this program contains vccc, a vector
println(vccc); //which is now available in the current program
```

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
```

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 test(int i,int j) {
            if (j>=i)
               return(true);
            return(false);
}
int s=10;
//We test through test
switch (s) with test {
            1:  println("1");
            2:  println("2");
            20:  println("20"); //This will be the selected occurrence
}
```

for(expressions;Boolean;nexts)

for is composed of three parts, an initialisation, a Boolean expression and a continuation part.

for (expression;boolean;next) {…}

You can use *continue or break to either go to the next element or to break in the middle of a loop.*

Example

```
for (i=0;i<10;i+=1) print("I=",i,"\n");
```

Multiple initializations and increments

Expressions, both in the initialization part and in the increment part, can contain more than one element. In that specific case, these elements should be separated by a comma.

Example

```
int i,j;

//Multiple initializations and multiple increments.
for (i=10,j=100;i>5;i--,j++)
        println(i,j);
```

for (var in container) {…}

This is a very specific sort of *for,* which is used to loop in a container, a string or a file.

Example

```
//we loop in a file
file f('myfile.txt',"r");
string s;

for (s in f)
    println(s);


//we loop in a vector of ints...
vector v=[1,2,3,4,5,6];
int i;

for (i in v)
    println(i);
```

while

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;
}
```

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.

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 1. *random()* returns a float value.

Example:

```
float rd=random();
```

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);
}
```

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);
        }
}
```

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.

Example with a variable

```
//We implement a trigger function
function react(self before,self after) {
        println("dico",before,after);
}
//which we associate with dico
map dico with react;

//whenever dico is modified, the "react" function is called
dico={'a':1,'b':6,'c':4,'d':6};
```

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. ***waitonjoin():*** 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*.

waitonjoin() 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: *waitonjoin()*.
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
   waitonjoin();
   println("End:"+x);
}

//we launch two of them
launch(0);
launch(1);
//and we wait for them to finish...
waitonjoin();
println("Termination");
```

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)**: *execute the system command s*
2. **createdirectory(string path):** *create a directory for the given path*
3. **vector v=listdirectory(string path):** *return the files in the directory at path*
4. **map info=fileinfo(string path):** *return a map with the following information for a given file: info["size"], info["date"] and info["pathname"].*
5. **realpath(string path):** *return the actual path for a given relative path.*
6. **getenv(string var):** *return the value of the environment variable:* var
7. **setenv(string var,string val):** *set the environment variable* var *with the value* val.

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

1. **gethostname():** *return the current host name. The socket does not need to be activated to get this information.*
2. **createserver(string hostname,int port,int nbclients):** *create a server on a host with a specific port.*
3. **createserver(int port,int nbclients):** *create a server on the local host with a specific port.*
4. **connect(string hostname,int port):** *connect a client to a specific host on a specific port.*
5. **int *clientid*=wait():** *the server wait for a client to connect. It returns the client identifier, which will be used to communicate with the client.*
6. **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.*

   Server Side
7. **read(clientid):** *read a KiF object on the socket with clientid*
8. **write(clientid,o1,o2...):** *write KiF objects on the socket with clientid.*
9. **receive(int clientid):** *read a simple string on the socket with clientid*
10. **send(int clientid,string s):** *write a simple string on the socket with clientid*
11. **close():** *close the socket*

12. **close(clientid):** *close the communication with clientid.*
13. **run(int client,string stopstring):** *put the server in run mode. Server can now accept Remote Method Invocation (RMI) mode.*

    Client Side
14. **read():** *read a KiF object on the socket*
15. **write(o1,o2...):** *write KiF objects on the socket*
16. **receive():** *read a simple string on the socket*
17. **send(string s):** *write a simple string on the socket*
18. **getframe(string name):** *return a frame object* remote *handle of name* name.
19. **getfunction(string name):** *return a function* remote *handle of name* name.
20. **close():** *close 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
        printlnerr("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
                        printlnerr("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. **open(string pathname):** *open a database*
2. **close():** *close a database*
3. **create(x1,x2,x3):** *create a table in a database, with the arguments x1,x2...*
4. **insert(table,column,value,...):** *insert a line in a table.*
5. **execute(string sqlcommand):** *execute a sql command.* 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.*
6. **begin():** *to enter the commit mode*
7. **commit():** *the SQL command are then processed. It should finish a series of commands initiated with a* begin.

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.execute("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();
```

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

a) **label(string txt)**: *set the label with a new text*
b) **labelsize(int i)**: *set or return the font size of the label*
c) **labelfont(int f)**: *set or return the font of the label*
d) **labelcolor(int c)**: *set or return the font color of the label*
e) **labeltype(int i)**: *set or return the font type of the label (see below for a description of the different types)*
f) **hide()**: *hide a widget*
g) **show()**: *show a widget*
h) **tooltip(string txt)**: *associate a widget with a tooltip*
i) **selectioncolor(int color)**: *set or return the widget selected color*
j) **coords()**: *return a vector of the widget coordinates*
k) **coords(int x,int y,int w,int h)**: *set the widget coordinates. It also accepts a vector instead of the four values.*

l) **backgroundcolor(int color):** *set or return the background color*

Label types

```
FL_NORMAL_LABEL
FL_NO_LABEL
FL_SHADOW_LABEL
FL_ENGRAVED_LABEL
FL_EMBOSSED_LABEL
```

wimage

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

m) **loadjpeg(string filename)**: *load a JPEG image*
n) **loadgif(string filename)**: *load a GIF image*

Utilization

Once a *wimage* 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. **create(int x,int y,int w, int h,string title)**: *Create a window without widgets, w and h are optional*
2. **begin(int x,int y,int w, int h,string title)**: *Create a window and begin initialisation, w and h are optional*
3. **end()**: *end creation*
4. **run()**: *Launch the GUI*
5. **close()**: *close the window*
6. **onclose(function,object):** *define a callback function to be called when the window is closed (see below)*
7. **menu(vector,int x,int y,int w, int h)**: *initialize a menu with its callback functions*

8. **resizable(object)**: *make the object resizable*
9. **sizerange(int minw,int minh, int maxw,int maxh)**: *define range in which the size of the window can evolve*
10. **modal(bool b)**: *If true make the window modal, with no parameter, it returns if the window is modal.*
11. **border(bool b)**: *If true add or remove borders. With no parameter return if the window has borders*
12. **drawcolor(int c)**: *set the color for the next drawings*
13. **rectangle(int x,int y,int w, int h, int c)**: *Draw a rectangle with optional color c*
14. **rectanglefill(int x,int y,int w, int h, int c)**: *Fill a rectangle with optional color c*
15. **lineshape(int type,int width)**: *Select the line shape and its thikness*
16. **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*
17. **line(int x,int y,int x1, int y1,int x2, int y2)**: *Draw a line between points, x2 and y2 are optional*
18. **arc(int x,int y,int x1, int y1, float a1, float a2)**: *Draw an arc*
19. **arc(float x,float y,float rad,float a1,float a2)**: *Draw an arc*
20. **pie(int x,int y,int x1, int y1, float a1, float a2)**: *Draw a pie*
21. **point(int x,int y)**: *Draw a pixel*
22. **circle(int x,int y,int r)**: *Draw a circle*
23. **textsize(string l)**: *Return a map with w and h as key to denote width and height of the string in pixels*
24. **drawtext(string l,int x,int y)**: *Put a text at position x,y*
25. **font(int f,int sz)**: *Set the font name and its size*
26. **redraw()**: Redraw the window
27. **rgbcolor(int color)**: *return a vector of the color decomposition of into RGB components*
28. **rgbcolor(int r,int g,int b)**: *return the* int *corresponding to the combination of RGB components.*
29. **rgbcolor(vector rgb)**: *return the* int *corresponding to the combination of RGB components, which are stored in a vector.*
30. **onmouse(int action, function,object)**: *Set the callback function on a mouse action with a given object as parameter*
31. **onkey(int action, function,object)**: *Set the callback function on a keyboard action with a given object as parameter*
32. **cursor(int cursortype,int color1, int color2)**: *Set the cursor shape. See below for a list of cursor shapes.*
33. **hide(bool v)**: *hide the window if v is true, show it otherwise*
34. **position(int x,int y)**: *position the window at the coordinates x,y*

35. **position():** *return a vector of the x,y position of the window*
36. **size(int x,int y,int w,int h):** *resize the window*
37. **size():** *return a 4 values vector of the window size*
38. **pushclip(int x,int y,int w, int h):** Insert a clip region, with the following coordinates
39. **popclip():** Release a clip region
40. **awake():** Awake a threaded window
41. **lock():** FLTK lock
42. **unlock():** FLTK unlock
43. **image(wimage image,int x, int y, int w, int h):** Display an image
44. **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.*
45. **alert(string msg)***: Pop up window to display an alert*

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);
```

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
```

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();
```

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, 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 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

Example

```
//we declare our window together with its associated drawing function and the object
function pushed(window w,string skey,int ikey,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);*

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

a) **i[a]**: *Extract character from the input at position a*
b) **i[a:b]**: *Extract characters between a and b*
c) **create(int x,int y,int w,int h,boolean multiline,string label)**: *Create an input area with* multiline if this parameter is true
d) **value()l(string v)**: *return the input buffer or set the initial buffer*
e) **insert(string s,int p)**: *insert s at position p in the input*
f) **selection()**: *return the selected text in the input*
g) **word(int pos)**: *return the word at position pos*
h) **color(int c)**: *set or return the text color*
i) **font(string s)**: *set or return the text font*
j) **fontsize(int c)**: *set or return the text font size*

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

a) **create(int x,int y,int w,int h,boolean multiline,string label)**: *Create an output area with* multiline if this parameter is true
b) **value(string v)**: *initialize the buffer*
c) **color(int c)**: *set or return the text color*
d) **font(string s)**: *set or return the text font*
e) **fontsize(int c)**: *set or return the text font size*

box (box definition)

This type is used to draw a box in the main window with some texts. It exposes the following methods:

Methods

a) **create(int x,int y,int w,int h, string label)**: *Create a box with a label*
b) **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

o) **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*
p) **when(int when1, int when2,...)**: *Type of event for a button which triggers the callback (see events below)*
q) **shortcut**(string keycode): *Set a shortcut to activate the button from the keyboard (see below for a list of shortcuts code)*
r) **color(int code)**: *Set the color of the button*
s) **value():** return the value of the current button
t) **align(int):** *define the button label alignment*
u) **image(wimage im,string label,int labelalign):** *Use the image as a button image*

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.

```
wimage 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. **value(int s)**: *set the choice initialization value*
3. **font(string s)**: *set or return the text font*
4. **fontsize(int c)**: *set or return the text font size*
5. **menu(vector s)**: *Initialize the menu. This should be the last operation in a wchoice creation.*

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",ca
llback_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();
```

table

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. **create(int x,int y,int w,int h,string label)**: *Create a table of objects, and starts adding*
2. **clear()**: *Clear the table*
3. **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.*
4. **cell(int R,int C):** *Return the value at row R and column C*
5. **row(int nb,int height)**: *Define the number of rows*
6. **row()**: *return the number of rows*
7. **rowheight(int height)**: *Define the row height in pixel*
8. **column(int nb)**: *Define the number of columns*
9. **column()**: *return the number of columns*
10. **columnwidth(int width)**: *Define the column width in pixel*
11. **font(int s)**: *set or return the text font*
12. **fontsize(int c)**: *set or return the text font size*
13. **colorbg(int c)**: *set or return the cell color background*
14. **colorfg(int c)**: *set or return the cell color foreground*
15. **when(string when)**: *Type of event to trigger the callback*
16. **selectioncolor(int color)**: *Color for the selected elements*
17. **boxtype(int boxtype)**: *box type*
18. **rowheader(int R,string label)**: *set the label of the row header for row R.*
19. **columnheader(int C,string label)**: *set the label of the column header for column C*
20. **rowheaderheight(int sz)**: *the size in pixel of the row header*
21. **columnheaderwidth(int sz)**: *the size in pixel of the column header*

Example

```
window w;
```

```
function callback_table(table x,map V,window w) {
        println(V);
}

table 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 define the number of rows, with their height in pixels
t.rowheight(20);
//we define the number of columns, with their width in pixels
t.columnwidth(60);
//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);
}
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

a) **e[a]**: *Extract character from the editor at position a*
b) **e[a:b]:** *Extract characters between a and b*
c) **create(int x,int y,int w,int h,string label)**: *Create an editor*
d) **selection()**: *return the selected text in the editor*
e) **cursorstyle(int style):** *set the cursor shape (see below)*
f) **addstyle(map styles):** *Initialize the styles for text chunks (see below for more details)*
g) **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)*
h) **getstyle(int start,int end):** *Return the style for each character of a chunk of text as a vector.*
i) **annotate(string s,string keystyle,bool matchcase):** *Each occurrence of* s *in the text is assigned the style* keystyle. m*atchcase is optional.*
j) **annotateregexip(string reg,string keystyle):** *Each string matching the xip regular expression reg is assigned the style keystyle.*
k) **highlight(int start,int end):** *highlight the characters between* start *and* end*.*
l) **highlight():** *return 1 or 0 if there is highlighted text in the editor. In the context of a string, returns the highlighted string*
m) **unhighlight():** *remove highlighting*
n) **cursor(int i):** *move the cursor to the* i^{th} *bytes.*
o) **cursor():***return the current position of the cursor in byte increments.*
p) **cursorchar():***return the current position of the cursor in character increments.*
q) **find(string s,int i,bool matchcase):** *find a string in the editor text, starting at position* i.
r) **rfind(string s,int i,bool matchcase):** *find a string in the editor text, starting at position* I, backward.
s) **load(string filename):** *load the content of a file into the editor.*
t) **save(string filename):** *save the content of the editor into a file.*

u) **value(string v)**: *return the text in the editor or initialize the editor*
v) **color(int c)**: *set or return the text color*
w) **font(string s)**: *set or return the text font*
x) **fontsize(int c)**: *set or return the text font size*
y) **word(int pos):** *return the word at position pos*
z) **line(int pos)**: *Return the line corresponding to pos or the line text itself. This line should be visible.*
aa) **line()**: *return the current line number or the line text itself*
bb) **linebounds(int pos***): return a vector with the start position and end position of the line at pos in bytes increments.*
cc) **linebounds()***: return a vector with the start position and end position of the current line in bytes increments.*
dd) **lineboundschar(int pos***): return a vector with the start position and end position of the line at pos in bytes increments.*
ee) **lineboundschar()***: return a vector with the start position and end position of the current line in character increments.*
ff) **gotoline(int l,bool highlight):** *goto line* **l** *and highlight it if true.*
gg) **onvscroll(function f, object o)**: *set the callback when scrolling vertically (see below for an example)*
hh) **onhscroll(function f, object o)**: *set the callback when scrolling horizontally (see below for an example)*
ii) **onmouse(int action,function f, object o)**: *set the callback when handling the mouse*
jj) **onkey(int action,function f, object o)**: *set the callback when scrolling vertically (see below for an example)*
kk) **append(string s):** *append a string at the end of the editor text*
ll) **insert(string s,int pos):** *insert a string at position pos*
mm) **byteposition(int pos)**: *convert a character position into a byte position (especially useful in UTF8 strings)*
nn) **charposition(int pos)**: *convert a byte position into a character position (especially useful in UTF8 strings)*

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

a) **create(int x,int y,int w,int h,string label)**: Create a scrolling region
b) **resize(object)**: make the object resizable

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

a) **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)*
b) **resize(object)**: *make the object resizable*
c) **bounds(int x,int y)**: *defines the slider boundaries*
d) **type(int x)**: *Value slider type (see below for the list of slider types)*
e) **align(int align)**: *define the slider alignement*
f) **step(int)**: *define the slider step*
g) **value()**:*return the slider value*
h) **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
```

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
```

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. **begin(int x,int y,int w, int h,string title)**: *Create a tab window and begin initialization*
2. **end()**: *end the* tabs *construction*
3. **add(wgroup g)**: *add dynamically a new tab.*
4. **remove(wgroup g)**: *remove the group g from the tabs*
5. **current()**: *return the current active tab*
6. **current(wgroup t)**: *activate this tab*

Group methods

The *group* object exposes the following methods:

1. **begin(int x,int y,int w, int h,string title)**: *Create a widget group and begin initialization*
2. **end()**: *end the* group *construction*
3. **activate()**: *activate the tab*

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 must be created with a callback function, whose signature is the following:

```
function callback_filebrowser(filebrowser f,myobject object);
```

It exposes the following methods.

Methods

a) **create(string intialdirectory,string filter,bool directory,string label)**: *Open a file browser, select directories if directory is true*
b) **value()**: *Return the selected file*
c) **close()**: *Close the file browser*

d) **ok()**: *return true if ok was pressed*

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',"*",true,"Choose your file");
```

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.

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.

- 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.

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 definition.

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 WINDOWS 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);* 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 KifTry method implementation.

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,
- KifVector,
- KifMap
- KiList.

For these objects, you MUST NOT USE "*new KifString*" or "*new KifVector*", but the following methods, which are exposed in KifCode (see kif.h):

```
Exported KifString* Providestring(string& z);
Exported KifString* Providestringraw(string z);
Exported KifInteger* Provideinteger(long val);
Exported KifFloat* Providefloat(double val);
Exported KifVector* Providevector();
Exported KifMap* Providemap();
Exported KifList* Providelist();
```

KiF, for efficiency reasons, maintains lists of predefined objects, which it actually reuses during execution. Thus, KiF has a list of maps, vectors or strings from which it extracts the right objects when needed. If you need to return a *string* object, then call *kifcode->ProvideString(value) at this effect.*

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);
```

Second, implement your *MethodCommand* in KifTry.

```
KifElement* MethodCommand(KifElement* contextualpattern,
            KifDomain* domain,
            KifCallFunction* callfunc) {
            if (callfunc->Size()!=1)
              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 *kifframetemplate.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 *kifframetemplate.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

kifframetemplate.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);
```

```
                }
            }
```