Le langage KIFF

Contenu

Résumé

Ce document décrit le langage KIFF sous sa forme française.

Quelques éléments

Un programme KIFF contient des déclarations de variables, de fonctions et de classes. Une variable peut être déclarée n'importe où, de même que les fonctions.

IMPORTANT

Les fichiers KIFF DOIVENT toujours être encodés en UTF8…

Commentaires

Les commentaires sont introduits avec les caractères : //

//Ceci est un commentaire

Pour transformer un bloc de lignes en commentaires, il faut utiliser :

/@ .. @/

/@

Ceci est un commentaire

Sur plusieurs lignes…

@/

Fonction

Une fonction est déclarée à l'aide du mot clef *fonction*, un nom et des paramètres.

Classe

Une classe est déclarée avec le mot clef *classe* suivi d'un nom. Une sous-classe est déclarée simplement à l'intérieur de sa classe parent.

Déclarations anticipées

Fonctions et classes peuvent être déclarées à l'avance avant l'implémentation décimalle.

Exemples

```
fonction toto(nombre i);
classe test;
```

Ramasse-miette : dictionnaire m=ramassemiette();

KIFF fournit une fonction qui retourne des informations à propos du ramasse-miette sous la forme d'un dictionnaire dont les clefs donnent des informations sur l'état de la mémoire. KIFF gère sept ramasse-miettes :

- Un ramasse-miette pour les chaines de caractères
- Un ramasse-miette pour les nombres
- Un ramasse-miette pour les décimaux
- Un ramasse-miette pour les tables
- Un ramasse-miette pour les dictionnaires
- Un ramasse-miette pour les listes
- Un ramasse-miette général

Fonction ramasse-miette : fonctionramassemiette

KIFF fournit un mécanisme simple pour détecter et vérifier quand le ramasse-miette est appelé. Si une fonction comme celle qui suit est implémentée en début de programme, elle sera systématiquement appelée en cas d'appel au ramasse-miette.

Exemple:

```
fonction ramasse() {
        afficheligneerr("Etat:",ramassemiette());
}

fonctionramassemiette(ramasse);
```

Passer des arguments à un programme KIFF

Un programme KIFF est généralement appelé avec une liste d'arguments qui sont dès lors accessibles via deux variables prédéfinies : *_args* et *_chemins.*

Exemple:

```
KIFF monprg c:\test\montexte.txt
```

Table système: _args

Cette variable conserve les arguments dans l'ordre où ils ont été fournis par l'utitisateur.

Exemple (d'après l'appel ci-dessus):

```
fichier f;
f.ouvreenlecture(_args[0]);
```

Chemins: _chemins

Cette table garde la trace des chemins de tous les programmes KIFF chargés en mémoire, y compris en position 0, le chemin du répertoire courant d'où KIFF a été lancé.

```
//Afficher tous les chemins en mémoire
itérateur it=_chemins;

pour (it.commence() tantque it.nfin() faire it.suivant())
        affiche("Chargé: ",it.valeur(),"\n");
```

Separateur de chemins: _sep

Les systèmes UNIX et Windows utitisent un séparateur différent, '/ ' sous UNIX et '\' sous Windows. Cette variable vous donne cette information.

Types de base

Chaque objet ou variable dans KIFF se doit d'être déclaré. KIFF fournit un jeu initial de types prédéfinis que l'utitisateur peut élargir grâce aux classes.

Types prédéfinis

La plupart des types suivants sont assez traditionnels dans les langages de programmation, bien qu'ici la terminologie ait été francisée.

Objets de base:

omni, chaine, nombre, décimal, long, bit, fraction, booléen, temps, appel, arbre

Objets complexes:

table, dictionnaire, fichier, itérateur

fonction

Une fonction se déclare à l'aide du mot clef *fonction*.

classe

Une *classe* est un type défini par l'utitisateur qui permet de conserver dans une structure commune valeurs et fonctions.

Déclaration de variables

On déclare une variable en fournissant d'abord son type, puis son nom, ou une suite de noms séparés par des virgules. On doit obligatoirement termnier cette déclaration par un « ; ».

Exemple:

```
//chaque variable peut être instanciée individuellement dans la liste
nombre i,j,k=10;
chaine s1="s1",s2="s2";
```

privée type nom;

Une variable privée ne peut pas être accessible en dehors de la classe où elle a été déclarée ou du fichier dans lequel elle a été déclarée. On

protège de cette façon l’accès à ces variables lors d’un partage de code par exemple.

Exemple

privée test toto;

Premier programme

Comme un exemple vaut dix mille mots, voici un premier petit programme qui affiche le contenu d'une chaine.

```
//déclaration
chaine s;
nombre i;

//Instanciation
s="abcd";
i=100;
//Affichage
afficheligne("S=",s," I=",i);
```

Exécution

S=abcd I=100

fonction, polynomiale, tâche

Une fonction doit être déclarée avec le mot clef *fonction* suivi de son nom et de ses paramètres. Une fonction peut retourner une valeur grâce à la fonction renvoie.
Veuillez noter que le type du retour n'est pas précisé dans la description de la fonction.
Il existe plusieurs types de fonctions :

Polynomiale

Une fonction polynomiale indexe le résultat d'un calcul effectué au sein de celle-ci sur la base de ses paramètres. Ainsi, le calcul n'est fait qu'une fois pour un jeu particulier de paramètres.

Exemple

```
polynomiale calcul(nombre j) {
        décimal k=j.cos()*j.sin()^10;
        renvoie(k);
}
```

Au premier appel de calcul(10), KIFF calculera cette valeur :

-8.39072 pour k.

Pour les appels suivants de calcul(10), le système utilisera la valeur déjà calculée pour la renvoyer, plutôt que de recommencer le calcul.

Note

Ce type de fonction construit un dictionnaire interne pour enregistrer ces valeurs. Il peut donc consommer des ressources et doit être utiliser de façon concertée.

tâche

Une tâche (*thread* en anglais) est une fonction qui s'exécute en parallèle du code principal.

Exemple:

```
tâche toto(nombre i) {
        i+=10;
}
```

```
toto(10);
```

protégée tâche

“protégée” permet d’éviter que deux tâches n’accèdent aux mêmes lignes de code en même temps. Une tâche protégée place un verrou au début de son lancement qui n’est relâché qu’à la fin de l’exécution. Cela peut s’avérer intéressant pour gérer des codes non réentrants.

Exemple

```
//Nous implémentons des tâches synchronisées
protégée tâche lancement(chaine n,nombre m) {
        nombre i;
        afficheligne(n);
        //Nous affichons nos valeurs
        pour (i=0;i<m;i++)
            affiche(i," ");
        afficheligne();
}

fonction principal() {
        //nous lançons notre tâche
        lancement("Premier",2);
        lancement("Second",4);
        afficheligne("Fin");
}

principal();
```

exécute:

```
Fin
Premier
0 1
Second
0 1 2
```

exclusive tâche

Une tâche exclusive ressemble par de nombreux points à une tâche protégée à une différence près. Tout d’abord exclusive n’a de sens qu’au sein d’une classe. Deuxièmement, à l’image de *synchronized* en Java, ce type de tâche bloque non pas l’accès à la méthode, mais à la classe elle-même. Ainsi, plusieurs tâches exclusives peuvent s’exécuter en parallèle, en revanche, une tâche exclusive pour un objet donné est protégée contre des appels concurrents. Ainsi, si l’objet O1 est partagé par des tâches différentes, et que sa méthode

T1 est une tâche. Alors O1.T1() ne pourra être appelé que d'une seule tâche à la fois.
Autrement dit , dans une tâche protégée, nous plaçons un verrou au sein de la fonction elle-même, alors que dans le cas d'une tâche exclusive, nous plaçons le verrou au sein de l'instance de classe.

exclusive tâche classmethode(..) {verrouille(instanceid)…}
protégée tâche méthode(…) {verrouille(méthodeid)…}

Exemple

```
//Cette classe expose deux méthodes
classe disp {

        //exclusive
        exclusive tâche eaffichage(chaine s) {
                afficheligne("Exclusive:",s);
        }

        //protégée
        protégée tâche paffichage(chaine s) {
                afficheligne("Protégée:",s);
        }
}

//Nous implantons une classe action
classe action {
        chaine s;
        //avec une instance "disp"
        disp cx;

        fonction _initiale(chaine x) {
                s=x;
        }

        //Appel avec protégée
        fonction paffichage() {
                cx.paffichage(s);
        }

        //Appel avec exclusive
        fonction eaffichage() {
                cx.eaffichage(s);
        }

        //Appel de l'isntance globale avec exclusive
        fonction affichage(disp c) {
                c.eaffichage(s);
        }
}

//Les instances communes
disp c;
```

```
table v;
nombre i;
chaine s="T";
pour (i=0;i<100;i++) {
        s="T"+i;
        action t(s);
        v.empile(t);
}

//Dans le premier cas, l'affichage sera ordonnée puisque le code protégé est non réentrant. //Un seul paffichage s'exécute à la fois
pour (i=0;i<100;i++)
        v[i].paffichage();

//Dans ce cas, l'affichage mélangera des eaffichages
pour (i=0;i<100;i++)
        v[i].eaffichage();

//Dans ce dernier cas, nous n'avons qu'un seul objet commun, ce qui induira de nouveau un //ré-ordonnement des sorties.
pour (i=0;i<100;i++)
        v[i].affichage(c);
```

joindre et attendsfinjoindre

Si le programme principal doit attendre la fin de l'exécution de toutes les tâches, il faut alors les déclarer en tant que « joindre » puis appeler dans le programme principal la fonction : attendsfinjoindre.

Plusieurs *attendsfinjoindre* peuvent fonctionner en parallèle s'ils sont eux-mêmes appelés depuis une tâche.

Operateur de flux: '<<<'

Quand vous lancez une tâche, le résultat de celle-ci ne peut généralement pas être enregistré dans une variable avec l'opérateur « = ». Les tâches vivent leur vie sans être liées de façon particulière avec le code appelant. Pour cette raison, KIFF fournit un opérateur particulier, dit opérateur de flux : <<<. Une variable de flux est une variable qui va être liée étroitement avec la tâche de telle façon que les valeurs retournées par celle-ci puissent s'enregistrer dans cette variable. La seule contrainte est que la vie de la variable soit au moins aussi longue que celle de la tâche. Une tâche ne peut évidemment pas sauvegarder une valeur dans une variable temporaire qui aura disparu avant même que la tâche ait fini son exécution.

Exemple

```
//Nous créons une tâche de type "joindre".
//Cette tâche retourne 2xi
joindre tâche Test(nombre i) {
    renvoie(i*2);
}
```

```
//Cette fonction lance 10 tâches
fonction Lance() {
    //Le dictionnaire pour enregistrer les valeurs
    dicotrién m;
    nombre i=0;
    //Nous lançons nos tâches et le résultat vient s'enregistrer dans m
    pour (i dans <0,10,1>)
        m[i]<<<Test(i);

    //Nous attendons la fin de nos tâches déclarées en "joindre" plus haut
    attendsfinjoindre();
    //Affichage du résultat
    afficheligne(m); //{0:0,1:2,2:4,3:6,4:8,5:10,6:12,7:14,8:16,9:18}
}

Lance();
```

Multiple définitions

KIFF permet de définir des fonctions dont la liste de paramètres diffère mais qui partagent le même nom. Dans ce cas, le contrôle d'arguments est plus exigeant que d'ordinaire.

Exemple:

```
fonction testmultipledeclaration(chaine s, chaine v) {
        afficheligne("Chaine:",s,v);
}

fonction testmultipledeclaration(nombre s, nombre v) {
        afficheligne("Nombre:",s,v);
}

//nous déclarons nos variables
nombre i;
nombre j;
chaine s1="s1";
chaine s2="s2";

//KIFF choisira la bonne fonction selon les arguments fournis
testmultipledeclaration(s1,s2); // implémentation avec chaine
testmultipledeclaration(i,j); // implémentation avec nombre
```

Important

A la différence du C++, KIFF ne considère pas l'ambiguité de sélection comme une erreur. Il choisit la première fonction qui correspond aux arguments de l'appel. L'ordre de déclaration est donc important.

Fonction : environnement()

Quand une fonction est appelée dans un certain contexte, dont la connaissance peut être utile, on peut appeler cette fonction pour

obtenir le type de la variable dans laquelle le résultat de la fonction sera rangé.

- Exemple:

```
//Notre fonction
fonction toto(nombre i) {
   chaine s=environnement();
   affiche("Environnement:",s,"\n");
   renvoie(i+10);
}

//Voici deux appels de fonction dans des contextes différents
nombre j=toto(10);
chaine s=toto(10);
```

Exécute

Environnement: nombre
Environnement: chaine

Nombre illimité d'arguments

Il est possible de déclarer une fonction avec un nombre illimité d'arguments. Il suffit alors de finir la description des arguments avec « … ». Ces arguments sont alors accessibles via une table dont le nom est celui de la fonction précédé d'un « _ ».

Exemple

```
//Une fonction avec un nombre illimité d'arguments
fonction test(nombre i,...) {
        //Les autres arguments sont stockés dans la table: _test
        afficheligne("Arguments:",_test);
        renvoie;
}

//Appel
afficheligne("Test:",test(14,18,90));
```

Exécute

Arguments: [18,90]

Attribut de fonction particulier: privée & stricte

Les fonctions peuvent être spécifiées avec deux attributs particuliers, placés avant le mot clef fonction: privée et stricte.

Note:

Si vous désirez utiliser les deux attributs dans la même déclaration : *privée* doit précéder *stricte*.

- privée [fonction | tâche | autoexécute | polynomiale]

Une fonction privée ne peut être accessible à l'extérieur d'une classe ou à l'extérieur d'un programme KIFF chargé depuis un autre programme.

- Exemple:

```
//Cette fonction est invisible pour les chargeurs externes…
privée fonction toto(nombre i) {
        i+=10;
        renvoie(i);
}
```

- stricte [fonction | tâche | polynomiale]

Par défaut, lorsque KIFF appelle une fonction avec des paramètres, Il peut parfois les convertir de façon implicite. Par exemple, un nombre peut être converti en une chaine. Le mot clef *stricte* force à une équivalence stricte entre les paramètres et les arguments.

Exemple:

```
stricte fonction teststrictdeclaration(nombre s, nombre v) {
        afficheligne("Nombre:",s,v);
}

chaine s1="s1";
chaine s2="s2";

//Dans ce cas, un message d'erreur sera renvoyé, car aucune fonction n'existe avec ces paramètres
teststrictdeclaration(s1,s2); //Pas d'implémentation en chaine
```

Classe

Une classe permet de déclarer dans la même structure des variables et des fonctions (appelées alors méthodes).

- Les variables membres sont instanciées au sein de la classe.
- Une méthode _initiale peut être déclarée qui sera automatiquement appelé lors de la création d'une instance de la classe.
- Les sous-classes sont déclarées dans le corps de la classe elle-même. Elles héritent automatiquement des méthodes et variables de la classe mère tout en pouvant les surcharger (les remplacer).
- Les fonctions et les variables peuvent être déclarées en tant que privée.
-

Exemple

```
classe maclasse  {
        nombre i=10;        //chaque nouvelle instance de la classe instanciera i avec
10
        chaine s="initial";

        fonction affichage() {
                affiche("Dans MACLASSE:"+s+"\n");
        }
        classe masousclasse  {
                fonction affichage() {
                        affiche("Dans MASOUSCLASSE:"+s+"\n");
                }
        }
}
```

Utiliser une classe

Une instance d'une classe se déclare en utitisant le nom de la classe comme type.

Exemple

```
maclasse premier; //première instance
mysousclasse subpremier; //crée une instance de sous-classe

//Nous créons une nouvelle instance
premier=maclasse; //équivalent à "new maclasse" dans C++ ou en Java

//Pour exécuter une méthode de  classe
```

```
maclasse.affichage();
```

fonction _initiale

La fonction _initiale est le constructeur de la classe. Elle est automatiquement appelée lors de la construction d'une nouvelle instance.
On transmet les paramètres à cette fonction lors de la déclaration de la variable de classe, en plaçant ces paramètres entre parenthèses devant le nom de la variable :

maclasse obj(i,j,k),

Il faudra évidemment qu'il y ait une fonction *_initiale* correspondante en nombre et en type de paramètres dans la classe.

Exemple

```
classe maclasse  {
        nombre i=10;
        chaine s="initial";

        fonction _initiale(nombre ij) {
                i=ij;
        }

        fonction affichage() {
                affiche("DANS MACLASSE:"+s+"\n");
        }
}

// Une nouvelle instance de maclasse est créée:
maclasse second(10); //Les paramètres sont passés à la fonction _initiale en les ajoutant au nom de la variable.
```

_finale

La fonction _finale est appelée lorsque le ramasse-miette détruit l'instance, ce qui généralement a lieu à la sortie d'une fonction au sein de laquelle l'instance a été déclarée.

Important

- Cette fonction n'a pas de paramètre.
- Un appel à cette fonction ne détruit pas l'instance.

Exemple

```
classe maclasse  {
        nombre i=10;        //chaque nouvelle classe instanciera i avec 10
```

```
	chaine s="initial";

	fonction _initiale(nombre ij) {
		i=ij;
	}

	fonction _finale() {
		affiche("DANS MACLASSE:"+s+"\n");
	}
}

nombre i=10;
tantque (i>=0) {
	//A la fin de chaque itération, la fonction _finale sera appelée.
	maclasse item(i);
	i--;
}
```

Ordre d'initialisation

Quand des variables sont déclarées au sein d'une classe F, l'appel à *_initiale* est faite de la classe mère vers ses enfants.

Exemple

```
//Nous déclarons deux classes
classe avecin {
	nombre i;

	//avec une fonction _initiale particulière
	fonction _initiale(nombre j) {
		i=j*2;
		afficheligne("avecin _initiale",i);
	}

}

//Cette classe déclare un objet de la classe "avecin"
classe test {
   nombre i;
//Nous déclarons une classe spécifique, dont la déclaration dépend de la variable i
  avecin w(i);

	//Notre fonction _initale pour cette classe...
	fonction _initiale(nombre k) {
		i=k;
		afficheligne("test _initiale",k);
	}
}

//Nous créons une instance t1 avec comme valeur initiale: 20
test t1(20);
```

Exécution

L'exécution donne les résultats suivants:

test _initiale 20
avecin _initiale 40

Comme nous le voyons sur cet exemple, la fonction _initiale de la classe test a été exécutée en premier. L'appel à la fonction _initiale de avecin a été effectué après l'exécution, permettant au système de prendre avantage de la valeur de i déclarée dans cette classe.

Exemple

```
//Cette classe déclare une classe spécifique "avecin"
classe test {
        nombre i;
        avecin w(i);

        //Notre fonction _initale pour cette classe...
        fonction _initiale(nombre k) {
                i=k;
                w=avecin(100);
                afficheligne("test _initiale",k);
        }
}

test t1(20);
```

Exécution

L'exécution donne les résultats suivants:

test _initiale 20
avecin _initiale 200

Comme nous le voyons sur cet exemple, l'initialisation explicite de « w » dans _initiale a remplacé la déclaration de "avecin w(i);", qui devient dès lors redondante.

- Création au sein du constructeur

Lorsque l'on instancie une variable dans _initiale, toute autre initialisation devient redondante.

Exemple:

```
classe test {
        nombre i;
        avecin w;
```

```
        fonction _initiale(nombre k) {
                i=k;
                w=avecin(100);
                afficheligne("test _initiale",k);
        }
}

test t1(20);
```

Important

Si des paramètres de constructeur sont requis pour « w » et qu'aucun n'est fourni ni dans la déclaration ni dans la classe, KIFF générera une erreur.

Faire référence à l'objet courant : *ça*

Il peut être nécessaire dans certains cas de faire référence à l'objet courant (le *this* du C++ ou de Java), en particulier si l'on veut donner en paramètre l'objet courant. On utilise le mot clef *ça* pour ce faire.

Exemple :

```
//Prédéclarion de notre méthode et de notre classe
classe app ;
fonction monapp(app e) ;

classe app {
    nombre i;

    fonction _initiale(nombre k) {
        i=k;
    }
    //la méthode afficher fait appel à la méthode monappel
    //qui exige un paramètre de type classe app…
    fonction afficher() {
        //ici fait référence à l'objet courant au sein duquel se fait l'appel
        monapp(ça);
    }
}

fonction monapp(app a) {
    afficheligne(a.i);
}

//On crée notre objet
app a(10);
//On affiche son résultat
a.afficher();
```

Variables communes

La déclaration de variables de classe se fait grâce au mot clef : *commune*. Ces variables deviennent dès lors disponibles pour toutes les instances de la classe.

Exemple

```
classe maclasse  {
 commune nombre i;     //Toutes les instances partagerons cette variable.
}

maclasse t1;
maclasse t2;
t1.i=10;
t2.i=15;
afficheligne(t1.i,t2.i);       //affiche la même valeur pour les deux variables : 15 15
```

Variables et fonctions privées

Certaines fonctions ou variables peuvent être déclarées comme privée dans une classe. Une fonction privée ou une variable privée peut uniquement être accédée depuis l'intérieur de la classe.

Exemple

```
classe maclasse  {
        nombre i=10;
        // variable privée
        privée chaine s="initial";

        fonction _initiale(nombre ij) {
                i=ij;
        }

        //privée fonction
        privée fonction modifie(nombre x) {
                i=x;
                s="Modifié avec:"+x;
        }

        fonction affichage() {
                modifie(1000); //Vous pouvez l'appeler d'ici
                affiche("IN MACLASSE:"+s+"\n");
        }
}

maclasse test;

//Appels illégaux
```

```
test.modifie(100);          //cette instruction est illégale car “modifie” est privée
afficheligne(test.s);       //Cette instruction est illégale car “s” est privée
```

Sous-Classer ou enrichir une classe

Il est possible dans KIFF de sous-classer ou même d’enrichir une classe existante. Une description de classe peut s’effectuer en quelques étapes simples, que l’on peut par la suite reprendre et enrichir à nouveau.

Enrichir

```
//Définition limitée d’une classe…
classe maclasse  {
        nombre i=10;
}

//On rajoute du code ici…
…

//Puis nous enrichissons notre classe
//Nous reprenons la déclaration précédente dans laquelle nous rajoutons notre nouveau code

classe maclasse  {
        fonction affichage() {
                afficheligne(i);
        }
}
```

Pré-déclaration de fonctions

Une fonction peut prédéclarée tandis que son corps sera défini plus tard.

```
classe maclasse  {
        nombre i=10;

        fonction affichage(); //nous préparons l’implémentation de la fonction affichage

}

//UN peu de code supplémentaire…
…

//Nous définissons alors le corps de la fonction

classe maclasse  {
        fonction affichage() { //Son implémentation effective…
                afficheligne(i);
        }
```

```
}
```

C'est particulièrement utile si deux classes doivent partager des appels croisés.

Sous-classes

```
…

//L'ajout d'une sous-classe…
classe maclasse  {
        //Nous rajoutons notre sous-classe…
        classe sousclasse {…}
}
```

Appel des définitions des classes supérieures: classe::fonction

Si nous avons besoin de faire appel à la définition d'une méthode de la classe parent au lieu de la méthode courante, on fait précéder le nom de cette méthode par le nom de la classe supérieure avec « :: » comme séparateur.

Exemple

```
//essai d'appel de sous-classes...

//Nous définissons une classe test, qui contient la sous-classe soustest
classe   test {
  nombre i;

  fonction _initiale(nombre k) {
            i=k;
  }

  fonction affichage() {
          afficheligne("Dans test",i);
  }

  classe soustest {
            chaine x;

            fonction affichage() {
                    afficheligne("Dans soustest",i);
                    test::affichage();//Nous appelons la définition de test
            }
  }
}

//Nous créons deux objets
test t(1);
subtest st(2);

//Nous appelons les différentes méthodes
t.affichage(); //affichage:"Dans test,1"
```

```
st.affichage();//affichage"Dans soustest,2" et "Dans test,2"
st.test::affichage(); //affichage "Dans test,2"
```

Conversion implicite

Il est possible de définir dans une classe des conversions implicites. Par exemple, pour définir une conversion implicite en « chaine », on peut déclarer une méthode « chaine » qui sera automatiquement appelé dès qu'un contexte de ce type sera nécessaire. Ces conversions sont aussi disponibles pour les nombres, les tables, les dictionnaires ou les décimaux.
Pour convertir en une autre classe, Il suffit de donner comme nom à cette fonction, celui de la classe pour laquelle on veut cette conversion.

Exemple:

```
classe maclasse  {
        nombre i=10;
        //privée variable
        chaine s="initial";

        fonction chaine() { //Une conversion en chaine de caractères.
            renvoie(s);
        }

        fonction nombre() { //Une conversion en nombre
          renvoie(i);
    }
}
```

Dans le cas d'une conversion en une autre classe, le nom de la fonction doit porter celui de cette classe.

Exemple

```
classe classeone  {
        nombre i=10;
}

classe classedeux  {
    nombre j=100;

    fonction classeone() { //Nous définissons notre conversion en classeone
            classeone f;
                f.i=j;
            renvoie(f);
    }
}
```

Grâce à cette conversion, Il deviendra possible de convertir un objet en un autre à la volée.

Fonctions de comparaison

Il est aussi possible d'enrichir une classe avec des méthodes de comparaison entre éléments de même classe. La fonction portera le même nom que l'opérateur qui lui correspond : ">","<","==","!=", "<=" et ">=".
Chaque fonction a un seul paramètre qui est comparée avec l'élément courant.
Voici une liste de ces fonctions:

1. égale: fonction ==(classe b);
2. différent: fonction !=(classe b);
3. inferieur: fonction <(classe b);
4. supérieur: fonction >(classe b);
5. inférieur ou égale: fonction <=(classe b);
6. supérieur ou égale: fonction >=(classe b);

Chacune de ces fonctions doit renvoyer vrai ou faux selon le résultat du test.

Exemple:

```
//implémentation de l'opérateur inférieur dans une classe
classe comp {
        nombre k;
        //nous implémentons l'opérateur inférieur
        fonction <(autre b) {
                si (k<b.k)
                    renvoie(vrai);
                renvoie(faux);
        }
}

//nous créons deux éléments
comp one;
comp deux;
//L'un vaut 10 et l'autre 20
one.k=10; deux.k=20;
//La méthode définie ci-dessus est alors appelée.
si (one < deux)
    afficheligne("OK");
```

Fonctions arithmétiques

De la même façon, les fonctions arithmétiques peuvent aussi être définies pour une classe donnée. Ces fonctions, à l'exception de ++ et de --, doivent en revanche avoir deux paramètres.
De plus, elles doivent renvoyer un élément de la même classe que ses arguments.

1. plus: fonction +(classe a, classe b);
2. moins: fonction -(classe a, classe b);
3. multiplier: fonction *(classe a, classe b);
4. diviser: fonction /(classe a, classe b);
5. puissance: fonction ^^(classe a, classe b);
6. décalage gauche: fonction <<(classe a, classe b);
7. décalage droit: fonction >>(classe a, classe b);
8. reste: fonction %(classe a, classe b);
9. ou binaire: fonction |(classe a,classe b);
10. xou binaire: fonction ^(classe a,classe b);
11. et binaire: fonction &(classe a,classe b);
12. "++": fonction ++();
13. "--": fonction --();

Exemple:

```
classe test {
	nombre k;

	fonction ++() {
		k++;
	}

	//Il est nécessaire de créer une nouvelle instance qui est renvoyée par la fonction
	fonction +(test a,test b) {
		test res;
		res.k=a.k+b.k;
		renvoie(res);
	}
}
test a,b,c;
c=a+b; //notre implémentation ci-dessus sera automatiquement appelée.
```

Intervalle et index

Il est aussi possible d'utiliser une classe comme table ou dictionnaire. On peut alors accéder à ces éléments via un index ou un intervalle. Il faut pour cela implémenter les fonction suivantes :

1. fonction [](omni idx,omni valeur): Cette fonction insère un élément dans une table à la position idx
2. fonction [](omni idx): Cette fonction renvoie le résultat à la position idx.
3. fonction [:](omni gauche,omni droite): Cette fonction renvoie les valeurs entre les positions gauche et droite.

Exemple:

```
classe myvect {
	table kj;

	//Cette fonction insère une valeur dans la table à la position idx
	fonction [](nombre idx,omni valeur) {
		kj[idx]=valeur;
	}

	//Cette fonction renvoie la valeur à la  position idx
	fonction [](nombre idx) {
		renvoie(kj[idx]);
	}

	//Cette fonction renvoie la valeur entre l et r.
	fonction [:](nombre l,nombre r) {
		renvoie(kj[l:r]);
	}

}

myvect test;
test[0]=10;	//nous appelons la fonction [](…)
test[1]=5;	// nous appelons la fonction [](…)

// nous appelons la fonction [:](…)
afficheligne(test[0],test[1],test[0:]);	//affiche: 10 5 [10,5]
```

Fonctions associées: AVEC opérateur

Une classe peut être déclarée avec une fonction associée qui est appelée chaque qu'une valeur de cette classe est modifiée. Il existe en fait trois cas :

1. La fonction associée est définie au niveau de la classe
2. La fonction associée est définie au niveau de la variable de classe
3. La fonction associée est définie au niveau d'un champ de la classe

Tout appel à une fonction associée (*dite aussi fonction de* rappel) déclarée au niveau de la classe est automatiquement supplantée par

une fonction associée déclarée au niveau de la variable, qui elle-même est supplantée par la fonction déclarée au niveau d'un champ de la classe.

Exemple

```
.
classe testcallavec;

fonction calllocal(testcallavec tx,nombre before,nombre after) {
	afficheligne("LOCAL",tx,before,after);
}

fonction callclasse(testcallavec tx,nombre before,nombre after) {
	afficheligne("CLASSE",tx,before,after);
}


//Nous déclarons une classe avec une fonction associée callclasse
classe testcallavec avec callclasse {
	//une fonction associée locale
	nombre i avec calllocal=10;
	nombre j=30;

	fonction chaine() {
		renvoie(i);
	}
}

fonction callvariable(testcallavec tx,nombre before,nombre after) {
	afficheligne("VARIABLE",tx,before,after);
}

//Cette variable est associée avec une fonction associée de variable
testcallavec callt avec callvariable;
testcallavec callt2;

callt.i=15; //Cette modification déclenchera calllocal associée avec i
callt2.i=20; //Ce modification déclenchera calllocal associée avec i
callt.j=15; //Ce modification déclenchera callvariable associée avec callt
callt2.j=20; //Ce modification déclenchera callclasse associée avec testcallavec dans la déclaration de la classe.
```

Exécution:

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

VARIABLE 15 30 15	//modification de callt.j
CLASSE 20 30 20		//modification de callt.j
```

Classe principale: _KIFFPRINCIPALE

La variable _KIFFPRINCIPALE est un cas particulier d'une instance de classe: c'est la variable principale de KIFF, celle qui détient le dictionnaire de toutes les fonctions et variables globales.

_KIFFPRINCIPALE peut s'utiliser pour créer des variables globales à la volée, ou détruire une instance particulière d'un objet en mémoire.

Indexation dans _KIFFPRINCIPALE

_KIFFPRINCIPALE fonctionne à la façon d'un dictionnaire dont la clef est le nom de la variable ou de la fonction globale.

Exemple:

```
//Nous créons une nouvelle instance:
nombre i;
_KIFFPRINCIPALE ["TOTO"]=i;
```

_KIFFPRINCIPALE .dépile("nom")

_KIFFPRINCIPALE exposes la méthode: "dépile", qui permet de virer une variable de la mémoire principale.

Exemple:

```
//Nous la détruisons:
_KIFFPRINCIPALE .dépile("TOTO");
```

Comportement:

- Comme dictionnaire, _KIFFPRINCIPALE renvoie un ensemble {clef:valeur…}, où clef est le nom de l'objet et valeur son type.
- Comme chaine, Il renvoie le dictionnaire ci-dessus sous sa forme en chaine de caractères.

Courtier en objet

_KIFFPRINCIPALE peut être utilisé pour créer des objets dans un serveur à la volée.

```
//Une classe test
classe test {
        nombre i;

        fonction _initiale(nombre v) {
            i=v;
```

```
        }

        fonction Valeur() {
                renvoie(i);
        }
}

//Cette fonction crée un nouvel élément global de classe: test
fonction création(chaine n,nombre i) {
        test t(i);
        _KIFFPRINCIPALE [n]=t;
}

//Nous créons cette nouvelle instance:
création("TOTO",10);
//Il est possible de faire référence à cette variable de la façon suivante:
afficheligne(_KIFFPRINCIPALE ["TOTO"].Valeur());//Affiche: 10
```

Extensions

Il est possible d'étendre certains types en leur rajoutant des méthodes spécifiques.

Cette notion d'extension est parallèle à celle de classe, à la différence près que son nom ne peut être que l'un des types suivants :

Types valides: ***chaine, automate, date, chrono, fichier, nombre, table, liste, dictionnaire, sac.***

La définition d'une extension est basée sur celle d'une classe. Si vous devez faire reference à l'élément courant, il faut utiliser une variable dont le nom commence par « _ » suivi du nom de l'extension.

Pour "extension table", la variable sera: *_table*.

Attention, si vous rajoutez des méthodes aux types dictionnaire ou table, tous les types apparentés seront aussi modifiés.

Exemple:

```
//Nous étendons le type dictionnaire
extension dictionnaire {

    //Nous ajoutons une méthode
    fonction renvoieETnettoie(nombre clef) {
        //Nous extrayons la valeur
        chaine s=_dictionnaire[clef];
        //que l'on retire du dictionnaire
        _dictionnaire.dépile(clef);
        renvoie(s);
    }
}


dictionnaire mx={1:2,3:4};

// renvoieETnettoie est maintenant disponible pour les types de dictionnaire.

chaine s=mx. renvoieETnettoie(1);

diconn imx={1:2,3:4};

nombre x=imx. renvoieETnettoie(1);
```

KIFF Contextuel

KIFF est un langage de programmation contextuel.

La façon dont une variable est interprétée dépend de son contexte. Ainsi lorsque deux variables sont utilisées avec un opérateur, le résultat de l'opération dépend du type de la variable de gauche, celle qui introduit l'opération ou qui reçoit le résultat de l'opération.

Exemple

Si nous déclarons deux variables, une chaine et un nombre, alors le "+" fonctionnera comme une concaténation ou comme une addition.

Par exemple dans ce cas, i est la variable de réception, faisant de cette opération une addition.

```
nombre i=10;
chaine s="12";
i=s+i;       //le s est converti en un nombre.
affiche("I="+i+"\n");

Exécute
 I=22
```

Dans l'autre cas, s est la variable de réception. L'opération est maintenant une concaténation:

```
nombre i=10;
chaine s="12";
s=s+i;       //le i est converti en une chaine.
affiche("S="+s+"\n");

Exécute
 S=1210
```

Conversion implicite

Cette notion de contexte est très importante, puisqu'elle définit la façon dont chaque variable doit être interprétée. Les conversions implicites sont automatiquement traitées pour un certain nombre de type de base. Par exemple, un nombre est automatiquement transformé en une chaine, avec comme valeur ses propres chiffres. Dans le cas d'une chaine, le contenu est transformé en un nombre si la chaine ne contient que des chiffres, sinon sa valeur est 0.

Pour des cas plus spécifiques, tel qu'une table ou un dictionnaire, alors les conversions implicites sont un peu plus complexes. Par exemple, une table comme nombre renvoie sa taille et comme chaine une représentation de cette table. Un fichier comme chaine renvoie son nom et comme nombre, sa taille en octets.

Conversion explicite

Dans le cas d'une classe, la conversion doit être explicitement fournie par l'utitisateur. Cela consiste à ajouter une fonction spécifique dont le nom correspond à l'un des types suivant: chaine, nombre, décimal, long, booléen, table ou dictionnaire.

```
classe maclasse  {
        nombre i=10;
        chaine s="initial";

        fonction _initiale(nombre ij) {
                i=ij;
        }

        fonction nombre() {
                renvoie(i);
        }
        fonction chaine() {
                renvoie(s);
        }
}


maclasse test(10);

//affiche convertit automatiquement chaque paramètre en une chaine
affiche("MACLASSE:",test,"\n");
```

Exécute
MACLASSE: initial

Fonctions prédéfinies

La plupart des objets dans KIFF viennent avec une liste de méthodes prédéfinies.

- Méthodes de base

Tous les types ci-dessous partagent les mêmes méthodes de base:

a) estun(typenom): vérifie si une variable a le type: typenom (comme chaine)

b) type(): renvoie le type d'une variable comme chaine.

c) méthodes(): renvoie la liste des méthodes disponible pour une variable selon son type.

d) infos(chaine nom): renvoie une aide à propos d'une méthode.

e) chaine(): renvoie l'interprétation d'une variable comme chaine

f) nombre(): renvoie l'interprétation d'une variable comme nombre

g) décimal(): renvoie l'interprétation d'une variable comme décimal

h) table(): renvoie l'interprétation d'une variable comme table

i) dictionnaire(): renvoie l'interprétation d'une variable comme dictionnaire

Objet omnipotent: *omni* (ou *auto*)

omni est un objet transparent, simitaire à un pointeur, qui ne requiert aucune transformation particulière quand un objet lui est affecté.

Note : On peut aussi utiliser le mot clef *auto* à la place de *omni.* Ceci en référence au type *auto* dans C++11, qui est utilisé de façon similaire dans KIFF.

- Exemple

```
fonction compare(omni x, omni y) {
        si (x.type()==y.type())
```

```
                affiche("C'est le même type d'objet");
}

//Par exemple, dans ce cas, la fonction compare reçoit deux paramètres dont les types peuvent varier.
chaine s1,s2;
compare(s1,s2);

//nous comparons deux classes
maclasse i1;
maclasse i2;
compare(i1,i2);
```

Expressions régulières à la KIFF

Il s'agit d'un type d'expressions régulières particulières à KIFF.

- Le méta-caractères

La liste des méta-caractères dans ces expressions est la suivante :

%d	pour n'importe quel chiffre
%p	pour toute ponctuation parmi: < > { } [ ]) , ; : . & | ! / \ = ~ # @ ^ ? + - * $ % ' _ ¬ £ €` “
%a	pour toute lettre
%c	pour toute minuscule
%C	pour toute majuscule
?	pour n'importe quel caractère
%?	pour le caractère “?” lui-même
%%	pour le caractère « % » lui-même

Exemple:

dog%c	correspond à dogs ou dogg
m%d	correspond à m0, m1,…,m9

- Les opérateurs *,+, () , ([])

Il est possible d'utiliser aussi les conventions de Kleene :

x*:	le caractère peut être répété 0 ou n fois
x+:	le caractère doit être présent au moins une fois
(x):	le caractère est optionel
([x,…,x]*,+):	définit un caractère qui peut avoir un ensemble de propriétés.
%+,%*	permet d'utiliser + et * comme caractère

- Exemples:

1) a*ed	correspond à aed, aaed, aaaed etc. le « a » peut être présent 0 ou n fois)
2) a%*ed	correspond à aed, a**ed, a***ed etc. n'importe quel caractères peut apparaître entre a et ed)

3) a%d*	correspond à a, a1, a23, a45, a765735 etc.
4) a[%d,%p]	correspond à a1, a/, un etc.
5) a[bef]	correspond à ab, ae ou af.
6) a[%d,bef]	correspond à a1, ab, ae, af, a0, a9 etc.
7) a[be]+	correspond à ab, ae, abb, abe, abbbe, aeeeb etc.

Type chaine, uchaine

Le type chaine (« string » *en anglais)* fournit un grand nombre de méthodes pour gérer les chaines de caractères. Il intègre aussi bien des méthodes d'extraction via des expressions régulières que d'autres méthodes de conversion.

uchaine est un type plus efficace pour manipuler de grosses chaines de caractères, car le système suppose un encodage unique pour l'ensemble des caractères. Le « u » signifie d'ailleurs « unicode » dans ce contexte.

- Méthodes

1. balisage(chaine o,chaine f): Segmente une chaine à la façon d'une chaine parenthétisée sur la base des chaines o et f.
2. capteerr(bool): capte ou relâche la sortie erreur
3. captestd(bool): capte ou relâche la sortie standard
4. cherche(chaine sub,nombre pos): Renvoie la position de la sous-chaine sub commençant à la position pos
5. compte(chaine sub,nombre db,nombre fn): Compte le nombre de sous-chaines sub entre db et fn.
6. dépile(): retire le dernier caractère
7. dernier(): renvoie le dernier caractère
8. désaccentue(): Retire les accents des lettres accentuées
9. dos() : convertit une chaine en encodage DOS
10. dosversutf8() : convertit une chaine DOS en UTF8
11. droit(nombre nb): renvoie les nb derniers caractères d'une chaine
12. éclate(chaine exploseur): explose une chaine selon exploseur et enregistre le résultat sous la forme d'une table de type *chaines*. Si exploseur est une chaine vide, alors celle-ci est explosée en une table de caractères

13. éclatev(chaine exploseur) : fonctionne de la même façon que éclate, mais conserve les chaines vides dans le résultat final. Ainsi, si une variable c contient “+T1++T2”: c.éclate(“+”) renvoie [“T1”,”T2] alors que c.éclatev(“+”) renvoie ["",“T1”,"",”T2].

14. éclatergx(rgx): Explose chaine avec l'expression regulière rgx. Renvoie une table de sous-chaines.

15. enmajuscule(): Teste si la chaine est uniquement en minuscule

16. enminuscule(): Teste si la chaine est uniquement en minuscule

17. estalpha(): Teste si une chaine ne contient que des caractères alphabétiques

18. estchiffre(): Teste si une chaine ne contient que des chiffres

19. estconsonne(): Teste si une chaine ne contient que des consonnes

20. estutf8(): Renvoie vrai si la chaine est encodée en UTF8

21. estvoyelle(): Teste si une chaine ne contient que des voyelles

22. évalue() : Remplace les méta-caractères HMTL par les caractères équivalents (voir plus bas).

23. extrait(nombre pos,chaine depuis,chaine c1,chaine c2...): extrait les sous-chaines entre 'depuis' et 'c1'...'cn' (la chaine la plus courte est utilisée). Renvoie une table de chaines

24. format(p1,p2,p3): Crée une chaine à partir du format enregistré dans la chaine courante, où chaque '%x' correspond à un paramètre, 'x' est la position de ce paramètre dans la liste des arguments. 'x' commence à 1.

25. gauche(nombre nb): renvoie les nb premiers caractères d'une chaine

26. html() : remplace les caractères Unicode par leur correspondant HTML.

27. insère(i,s): insère la chaine s en position 'i'

28. latin(): convertit une chaine UTF8 en LATIN

29. levenshtein(chaine c): Renvoie la distance d'édition avec 'c' selon l'algorithme de Levenshtein.

30. saisie(): Lit une chaine depuis le clavier

31. lisp() : relit une structure parenthétisée (voir ci-dessous)

32. lisp(chaine o,chaine f) : relit une structure enchassée basées sur *o* et *f* comme séparateur.

33. majuscule(): Met la chaine en majuscule

34. milieu(nombre pos,nombre nb): renvoie les nb caractères commençant à la position pos d'une chaine

35. minuscule(): Met la chaine en minuscule

36. octets(): Renvoie la chaine comme table d'octets

37. ord(): renvoie le code ASCII du caractère en tête, ou une liste de tous les codes ASCII si le receveur est une table

38. positioncarac(nombre pos): convertit une position octet en une position caractère

39. positionoctets(nombre pos): convertit une position caractère en une position octet (utiles avec les chaines UTF8)

40. rcherche(chaine sub,nombre pos): Renvoie la position de la sous-chaine sub par l'arrière commençant à la position pos

41. regex(rgx): Renvoie les sous-chaines correspondant à rgx

42. regexip(rgx): Renvoie les sous-chaines correspondant à rgx

43. remplace(sub,str): Remplace la sous-chaine correspondant à sub avec str

44. remplacergx(rgx,str): Remplace la sous-chaine correspondant à rgx avec str

45. remplit(nombre nb,chaine c): crée une chaine contenant nb caractères ***c***

46. renverse(): inverse la chaine

47. retiredernier(nombre nb): retire les nb derniers caractères d'une chaine

48. retirepremier(nombre nb): retire les nb premiers caractères d'une chaine

49. rogne(): retire les caractères d'espace

50. rognedroit(): retire les caractères d'espace à droite

51. rognegauche(): retire les caractères d'espace à gauche

52. segmente(bool virgule, bool séparateur) : segmente une chaine selon les espaces et les ponctuations. Le traitement des nombres est contrôle par virgule qui vaut « faux » si l'on utilise le « . » comme séparateur décimal ou « vrai » pour la virgule. Le séparateur indique si les milliers au sein d'un nombre sont contrôlés par une ponctuation : 12.300,34 sera par exemple reconnu avec *segmente(vrai,vrai)*.

53. utf8(): convertit une chaine LATIN en UTF8

Méta-caractères

Si vous utilisez des chaines déclarées entre "", alors KIFF reconnaitra automatiquement les méta-caractères suivants :

- \n, \r et \t qui sont respectivement, le retour à la ligne, le retour chariot et la tabulation.

KIFF reconnait aussi un autre ensemble plus vaste de méta-caractères, qui seront automatiquement traduit pour vue via la méthode: évalue().

- Code décimal: \ddd, lequel est traduit par le caractère Unicode correspondant: \048 est par exemple '0'.

- Code hexadécimal: \xhh, lequel est traduit par le caractère Unicode correspondant: \x30 est le caractère '0'.

- Code Unicode: \uhhhh, lequel est traduit par le caractère Unicode correspondant: \u0030 est le caractère '0'.

- &#d(d)(d)(d); lequel est traduit par le caractère Unicode correspondant: &#30; est le caractère '0'. Cet encodage apparait dans les documents XML ou HTML.

- &nomcode; pour lesquelles existe une longue liste de caractères equivalents (XML et HTML). Ainsi: &eaccute; est le caractère: é.

De façon inverse, la méthode "html" renvoie une chaines dans laquelle les caractères Unicode sont traduits par leur forme HTML.

Opérateurs

sub *dans* s: teste si sub est une sous-chaine de s

pour (c dans s) {…}: itère parmi tous les caractères. A chaque itération, c contient un caractère de s.

+: concatène deux chaines.

“…”: définit une chaine, où les méta-caractères tels que “\n”,”\t”,”\r”,”\”” sont interprétés.

‘…’: définit une chaine, où les méta-caractères ne sont pas interpretés. Cette chaine ne peut contenir le caractère “’”.

Indexes

str[i]: renvoie le $i^{ème}$ caractère d’une chaine

str[i:j]: renvoie la sous-chaine entre i et j. i et j peuvent être soit des indexes soit des chaines, que le système utilisera pour extraire la sous-chaine. Notez que si j n’est pas précisé, alors la chaine sera extraite jusqu’à la fin.

Les indexes négatifs sont calculés comme déplacement à partir de la fin de la chaine.

Il est aussi possible d’extraire une sous-chaine en précisant la chaine à rechercher en lieu et place des indexes. Dans ce cas, si la chaine est précédé d’un « - », la recherche se fait à partir de la fin de la chaine. De plus, si l’on utilise des indices sous la forme de chaine conjointement avec des indexes numériques positifs, on considère alors ces indexes numériques comme étant un nombre de caractères après la chaine.

Enfin, il est possible d’utiliser le domaine isolé par des indexes pour remplacer une sous-chaine par une autre.

Exemples

```
chaine c="Il a passé l’été en Bretagne.";

c[12]       é                   //position absolue
c[-12]      e                   //position absolue
c[10:16]    l’été               //positions absolues
c[10:]      l’été en Bretagne.  //positions absolues
c["été":7]  été en Bre          //positions à partir de la chaine trouvée
```

c["été":-4]	**été en Breta**	**//positions à partir de la fin de la chaine**
c[-"a":]	**agne.**	**//recherche de la dernière instance de 'a'**
c[-"a":]="#"	**Il a passé l'été en Bret#**	**//Remplacement de la sous-chaine…**

- Comme nombre ou comme décimal

Si la chaine contient des chiffres, alors elle est convertie dans le nombre équivalent, sinon sa conversion est 0.

- lisp() ou lisp(chaine ouvre, chaine ferme)

KIFF fournit aussi une façon très commode de relire des expressions parenthétisées pour les transformer en table automatiquement.

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

KIFF fournit donc une method: *lisp* qui prend une structure telle que celle-ci-dessus et la transforme directement en *table*.

table t=c.lisp(); //c contient une expression telle que ci-dessus

Il existe une autre variation de cette méthode qui prend en entrée les caractères ouvrants et fermants de l'expression à analyser.

Exemple:

KIFF peut ainsi analyser la chaine suivante:

< <S <NP-SBJ They>

<VP make

<NP the argument>

<PP-LOC in

<NP <NP letters>

<PP to

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

Avec l'instruction suivante:

```
table t=s.lisp('<','>');
```

balisage(chaine ouvre, chaine ferme)

balisage est similaire à la méthode *lisp,* mais au lieu d'utiliser des caractères, elle utilise des chaines. *Il est déconseillé d'utiliser cette méthode pour des fichiers xml, il est préférable d'utiliser* xmldoc *plutôt.*

```
chaine s="OUVRE Ceci est OUVRE un bel OUVRE exemple FERME FERME FERME";
table t=s.balisage('OUVRE,'FERME);
```

Sortie: t=[['Ceci', 'est', ['un','bel', ['exemple']]];

Divers exemples de manipulation de chaines de caractère

```
//Quelques manipulations de chaines
chaine s;
chaine x;
chaines v;

//Simples manipulations
s="12345678a";
x=s[0];                 // valeur=1
x=s[2:3];                       // valeur=3
x=s[2:-2];              //valeur=34567
x=s[3:];                //valeur=45678a
x=s[:"56"];             //valeur=1234
x=s["2":"a"];           //valeur=2345678
s[2]="ef";                      //valeur=12EF45678a

//Les 3 derniers caractères
x=s.droit(3);           //valeur=78a

//Une explosion selon les espaces
s="a b c";
v=s.éclate(" ");        //v=["a","b","c"]
//Equivalent et un peu plus vite
s.éclate(" ",v);        //v=["a","b","c"]


//regex, x est une chaine, nous cherchons la première corespondance
x=s.regexip("%d%d%c");          //valeur=78a

//Nous avons un patron lselon lequel nous explosons notre chaine
```

```
s='12a23s45e';
v=s.regexip("%d%d%c");                    // valeur=['12a','23s','45e']
x=s.remplaceregexip("%d%ds","X");         //valeur=12aX45e

x=s.remplaceregexip("%d%1s","%1");        //valeur=12a2345e

//Expressions régulières
chaine rgx='\w+day';
chaine str="Yooo Wesdenesday Saturday";
table vrgx=str.regex(rgx);                //['Wesdenesday','Saturday']
chaine s=str.regex(rgx); //Wesdenesday
nombre i=str.regex(rgx);// position est 5

//Nous utilisons (…) pour isoler des éléments qui seront stockés dans la table
rgx='(\d{1,3}):(\d{1,3}):(\d{1,3}):(\d{1,3})';
str='1:22:33:444';
vrgx=str.éclatergx(rgx);  // [1,22,33,444]

str='1:22:33:4444';
vrgx=str.éclatergx(rgx);  //[] (4444 contient 4 chiffres)


str="A_bcde";
si (str.regex('[a-zA-Z]_.+'))             //Correspondance complète
        afficheligne("Yooo");             //Yooo

str="ab(kiff12,kiff14,kiff15,kiff16)";
tablechaines v=str.extrait(0,"kiff",",",")"); //Résultat: ['12', 14',' 15',' 16']

chaine frm="Ceci %1 est un %2 de %1 avec %3";

chaine s=frm.format("tst",12,14);
afficheligne(s); //Résultat : Ceci tst est un 12 de tst avec 14
```

Type: octet, court, nombre, décimal, long

KIFF fournit plusieurs types numériques : octet, court, nombre, long, décimal et fraction, qui sont décrites dans les sections suivantes.

Note à propos de l'implémentation C++:

Nombre et décimal *(int et float en anglais)* ont été implémentés sous la forme respectivement d'un long et d'un double. long est un nombre sur 64 bits, soit un __int64 sur Windows ou "long long" sur Unix.

- Méthodes:

1. #(): renvoie le complément de bits
2. atan(): arc tangent
3. bit(nombre ième) : renvoie *vrai* si le ième bit est à 1.
4. octets() : renvoie la représentation sous-jacente du nombre.
5. car(): renvoie le caractère ascii correspondant à ce nombre.
6. cos(): cosinus
7. exp():Exponentiel
8. format(chaine forme): renvoie une chaine formatée selon le patron dans *forme*. (ce format est le même que le format de sprintf dans C++)
9. facteurs() : renvoie sous la forme d'une tablenombres, la décomposition en nombres premiers.
10. fraction(): renvoie la valeur comme une fraction.
11. lire(): Lit un nombre depuis le clavier
12. ln(): log népérien
13. log(): log en base 10
14. sin(): sinus
15. racar(): racine carré
16. tan(): tangente

Hexadécimal

Un nombre hexadécimal commence toujours avec “0x”. Une déclaration hexadécimale peut mélanger les majuscules et minuscules parmi les chiffres hexa suivants: A,B,C,D,E,F.

Opérateurs

+,-,*,/:	opérateurs mathématiques
<<,>>,&,|,^:	opérateurs sur les bits
%:	division modulo
^^:	puissance (2^^2=4)
+=,-= etc:	opérateurs intégrant la valeur de la variable de réception

Exemple

```
décimal f;
nombre i=10;
nombre j=0xAb45;        //nombre Hexadécimal

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

Type bit

Le type bit implémente une table de bits, qui peut être utilisé pour enregistrer des valeurs Booléennes. Une table de bits peut être transformé en une table ou un dictionnaire de nombres. Il peut aussi être itéré. Il peut aussi prendre comme valeur, un nombre, un décimal ou un long.

Note:

Quand on construit une table de bits à partir d'un nombre, d'un décimal ou d'une fraction, la représentation binaire dépend de la plate-forme et le résultat peut s'avérer différent d'une machine à l'autre.

Méthodes

1. #(valeur): renvoie le complément de la table de bits

2. bit vb(nbbits): crée une table de bits de taille nbbits. Nnbits définit la taille en bits de la table de bits. Cependant, comme les bits sont enregistrés dans des entiers de 16 bits, la taille décimalle peut être supérieure à celle choisie. Par exemple, bit v(25) générera une table de 32 bits.

Opérateurs

<<,>>,&,|,^: opérateurs sur les bits. L'opérateur « << » (décalage gauche) peut dans certains cas augmenter la taille de votre table de bits.

Comme chaine

Il renvoie une représentation hexadécimale de la table de bits.

Comme table ou comme dictionnaire

Il renvoie une table ou un dictionnaire de nombres

Comme nombre ou comme décimal

Il renvoie la transformation en un nombre des 32 premiers bits.

Exemple

```
bit test1(62); //nous créons deux table de bits
bit test2(64);
```

```
test1=234; //nous initialisons le premier avec 234
test2=654531;
nombre i=test2;  //nous transformons cette table de bits en un nombre…
table v=test1; //nous transformons notre  table de bits en une table de nombre.
test1=test1 & test2; //nous calculons le « et binaire » entre test1 et test2
test1<<=7; //décalage gauche
test1=#(test1); // le complément de la table de bits
test1[1]=vrai; //modification du deuxième bit de la table de bits
afficheligne(test1[0],test1[1],test1[2],test1[3]);
```

Type bits (représentation creuse des bits)

Le type bits implémente un dictionnaire de bits, pour enregistrer un grand nombre de valeurs Booléennes. Un dictionnaire de bits peut être transformé en un dictionnaire de nombres. Il peut être itéré. Il peut aussi prendre comme valeur un nombre, un décimal ou un long.
Le type bits implémente une représentation creuse des bits, à la différence du type bit qui implémente une table de tous les bits. Ils sont de plus enregistrés sur des entiers de 64 bits et non de 16 bits.

Exemple:

Si nous déclarons les deux variables suivantes:

```
bit vbit;
bits mbit;

vbit[120]=1; //Nous créons 120/16=8 short pour enregistrer tous les bits nécessaires. Le dernier short aura le 8ième bit à 1.

mbit[120]=1 //Nous créons juste un élément, dont la clef est 1 (=120/64), et dont le bit 56 est à 1.
```

Méthodes

1. #(valeur): renvoie le complément de la table de bits

Opérateurs

&,|,^: opérateurs sur les bits.

Comme chaine

Il renvoie une représentation hexadécimale de la table de bits.

Comme dictionnaire

Il renvoie un dictionnaire de nombres.

Comme nombre ou comme décimal

Il renvoie la transformation en un nombre des 32 premiers bits.

Exemple

```
bits test1(62); //nous créons deux table de bits
bits test2(64);
test1=234; //nous initialisons le premier avec 234
test2=654531;
```

```
nombre i=test2;  //nous transformons cette table de bits en un nombre…
dnombres m=test1; //nous transformons notre table de bits en un dictionnaire de nombre.
test1=test1 & test2; // »et binaire » entre test1 et test2
test1=#(test1); // complément de notre table de bits
test1[1]=vrai; //modification du deuxième bit de notre table de bits
afficheligne(test1[0],test1[1],test1[2],test1[3]);
```

Type fraction

Le type fraction est une façon de conserver une information numérique sans perte. Ce type peut être utilisé de façon transparente dans tous les calculs avec des décimaux ou des entiers. Il dispose de quelques méthodes particulières.

- Méthodes:

    1. d(): renvoie le dénominateur de la fraction
    2. d(nombre v): initialise le dénominateur de la fraction
    3. fraction f(nombre n,nombre d): une fraction peut être créée en fournissant un numérateur et un dénominateur. Par défaut, le numérateur est 0 et le dénominateur est 1.
    4. inverse(): inverse le dénominateur avec le numérateur
    5. n(): renvoie le numérateur de la fraction
    6. n(nombre v): initialise le numérateur de la fraction
    7. nd(nombre n,nombre d): initialise le numérateur et le dénominateur de la fraction

- Comme chaine, comme nombre ou comme décimal

KIFF automatiquement crée le décimal ou le nombre approprié, en calculant la valeur de la fraction. Cette traduction introduit généralement une perte de précision. De plus à chaque étape, la fraction est systématiquement simplifiée.

Comme chaine, KIFF renvoie: “NUM/DEN”

Exemples:

```
//nous créons deux fractions
fraction f(10,3);
fraction g(18,10);
//Nous ajoutons g à f.
f+=g;
afficheligne(f); //affichage: 77/15
```

Type table

Une table est un conteneur qui enregistre les informations dans une table de taille infinie.

- Méthodes

1. applique(a,b,c): applique sur chaque élément une fonction, ou applique toutes les fonctions dans le conteneur
2. ajouteattentes(t1,t2,...,tn): Ajoute de nouvelles tâches à la liste des tâches du tableau (voir *attentes*, *(uniquement disponible pour le type* tableau).
3. attentes(t1,t2,...,tn,p1,p2,..pn): Lance un ensemble de tâches en attente de déclenchement, avec tableau comme premier argument de chaque appel. p1,p2,...pn sont des paramètres optionels passés en arguments lors de l'appel de chaque tâche *(uniquement disponible pour le type* tableau).
4. déclenche(bool réinit): Déclenche les tâches, si *réinit* est vraie, alors réactive les tâches. Voir plus loin pour un exemple. *(uniquement disponible pour le type* tableau).
5. dépile(): retire le dernier élément de la table.
6. dernier(): renvoie le dernier élément de la table
7. empile(a): rajoute en queue de table
8. insère(i,x): insère l'élément x à la position i
9. intervalle(premier,dernier,pas) : *pas* est optionnel, s'il est omis, il vaut 1. Crée une table contenant tous les éléments entre premier et dernier par pas de *pas.* Le type des éléments dans la table dépend du type de *premier* et de *dernier.*
10. joins(chaine sep): concatène chaque élément dans la table en une chaine ou chaque élément est séparé des autres par sep
11. octets(): Renvoie la chaine correspondant à une table d'octets. Chaque valeur doit être comprise entre 0..255
12. produit(): Multiplie chaque élément avec les autres
13. raz(): nettoie la table

14. renverse(): renverse l'ordre des éléments dans la table

15. somme(): Additionne chaque élément avec les autres

16. teste(i): teste si i est une position valide dans la table

17. texte(): Renvoie la chaine correspondant à une table d'octets. Chaque valeur doit être comprise entre 0..255

18. trie(compare): Trie le contenu du table selon une fonction de comparaison, laquelle est optionnelle

19. triechaine(bool ordre): trie le contenu de la table dont chaque élément est considéré comme une chaine. *ordre=faux* le tri est croissant*, ordre=vrai* le tri est décroissant*.*

20. trielong(bool ordre): trie le contenu de la table dont chaque élément est considéré comme un long. *ordre=faux* le tri est croissant*, ordre=vrai* le tri est décroissant*.*

21. trienombre(bool ordre): trie le contenu de la table dont chaque élément est considéré comme un nombre. *ordre=faux* le tri est croissant*, ordre=vrai* le tri est décroissant*.*

22. triedécimal(bool ordre): trie le contenu de la table dont chaque élément est considéré comme un décimal. *ordre=faux* le tri est croissant*, ordre=vrai* le tri est décroissant*.*

Initialisation

Une table peut être initialisé avec une structure entre “[]”.

```
table v=[1,2,3,4,5];
table vs=["a","b","v"];
table vr=intervalle(10,20,2); // vr est initialisé avec [10,12,14,16,18];
vs= intervalle (‘a’,’z’,2); //vr est initialisé avec ['a','c','e','g','i','k','m','o','q','s','u','w','y']
```

Opérateurs

x dans vect: renvoie vrai ou une liste d’indexes, selon la variable de réception. Si la table contient des chaines, alors la comparaison se fait sur une égalité stricte entre les chaines. Le système ne descend pas récursivement pour effectuer un « dans » sur les chaines présentes dans le conteneur. Pour obtenir ce comportement, il faut utiliser une fonction de comparaison (voir plus bas).

pour (s dans vect) {…}: itère parmi les valeurs. A chaque itération “s” contient une valeur de vect.

+,*,-,/ etc..: ajoute etc.. une valeur à chaque élément d'une table ou ajoute chaque valeur d'une table à un autre.

&,|: intersection ou union de deux tables

- Comme nombre ou comme décimal

Il renvoie la taille de la table

- Comme chaine

Il renvoie une structure, où chaque élément est séparé des autres par une virgule.

- **Extraire des variables d'une table: Patron de table**

Un patron de table permet d'initialiser des variables en les déclarant sous une forme qui correspond aux éléments de cette table. Un patron se divise en une partie qui contient aussi bien des variables que des valeurs atomiques et une queue introduite par l'opérateur "|".

On peut les utiliser de deux façons:

- Dans un assignement:
    a) [a,b|v]=[1,2,3,4,5], alors a=1, b=2 and v=[3,4,5]
- Dans une boucle *pour ..dans*
    a) pour ([a,b|v] dans [[1,2,3,4],[3,4,5]]) etc…

       première itération, a=1,b=2 and v=[3,4]

       seconde itération, a=3,b=4 and v=[5]

- Indexes

vect[i]: renvoie le $i^{ième}$ caractère d'une table

vect[i:j]: renvoie le sous-table entre i et j.

- Exemple

```
table vect;

vect=[1,2,3,4,5];
affiche(vect[0]);          //affichage: 1
affiche(vect[0:3]);              //affichage: [1,2,3]
```

```
vect.empile(6);
affiche(vect);              //affichage: [1,2,3,4,5,6]
vect.dépile(1);
affiche(vect);              //affichage: [1,3,4,5,6]
vect=vect.reverse();
affiche(vect);              //affichage:[6,5,4,3,1]
vect.dépile();
affiche(vect);              // affichage:[6,5,4,3]
vect+=10;
affiche(vect);              // affichage:[16,15,14,13]
```

- Exemple (trier des nombres dans une table)

//Cette fonction doit renvoyer vrai ou faux
//Le type des paramètres détermine le sens de la comparaison

```
fonction compare(nombre i,nombre j) {
        si (i<j)
            renvoie(vrai);
        renvoie(faux);
}

table myvect;
itérateur it;
myvect=[10,5,20];
myvect.tri(compare);
it=myvect;
pour (it.commence() tantque it.nfin() faire it.suivant())
   affiche("Contenu:"+it.clef()+"="+it.valeur(),"\n");
```

```
EXÉCUTE
Contenu:0=5
Contenu:1=10
Contenu:2=20
```

- Exemple (trier des nombres dans une table mais vus comme des chaines)

//Cette fonction renvoie vrai ou faux
//Le type de le paramètres déterminera son comportement, dans ce cas, chaque élément sera soit une chaine ou soit converti en chaine.

```
fonction compare(chaine i,chaine j) {
        si (i<j)
            renvoie(vrai);
        renvoie(faux);
}

table myvect;
itérateur it;
myvect=[10,5,20];
myvect.tri(compare);
```

```
it=myvect;
pour (it.commence() tantque it.nfin() faire it.suivant())
        affiche("Contenu:"+it.clef()+"="+it.valeur(),"\n");

EXÉCUTE (Ce temps nous sort out chaines)
Contenu:0=10
Contenu:1=20
Contenu:2=5
```

Exemple: modification de chaque élément d'une table avec une fonction

Il est possible dans KIFF d'appeler *applique* avec comme premier paramètre une fonction ou une variable d'appel. Ainsi, chaque élément de la table peut être appelé et modifié si nécessaire.

```
//Nous implémentons une méthode, avec le premier élément extrait de la table
fonction modifie(omni tableÉlément,chaine s) {
        afficheligne(s, tableÉlément);
//nous le modifions...: un élément omni est équivalent à un pointeur passé comme paramètre
        tableÉlément+=1;
}

//Nous créons une table, avec des valeurs numériques
table v=[1,2,3,4,5];
//Nous appliquons notre fonction à chaque élément
v.applique(modifie,"Modification");
//nous affichons
afficheligne("Nouveau:",v);

EXÉCUTE:
//modifie a été appelé 5 fois, d'où le résultat ci-dessous
Modification 1
Modification 2
Modification 3
Modification 4
Modification 5
Nouveau: [2,3,4,5,6] //Chaque valeur a été modifiée
```

Calcul d'une factorielle

```
//x provient de la table et d est une valeur fournie à applique comme paramètre
//Elles sont déclarées comme "omni" pour permettre leur modification
fonction fact(omni x,omni d) {
    x*=d;
    d=x;
}

nombre d=1;
tablenombres iv=[1,2,3,4,5,6];
iv.applique(fact,d); //iv=[1,2,6,24,120,720]
```

Exemple d'utilisation de attentes/déclenche

```
//Nous déclarons un ensemble de tâches.
tâche f1(tableau t,table res,chaine s) {
    si (t[0]==1)
        afficheligne(1,t);
    sinon
        res.empile(1);
}

tâche f2(tableau t,table res,chaine s) {
    si (t[0]==2)
        afficheligne(2,t);
    sinon
        res.empile(2);
}

tâche f3(tableau t,table res,chaine s) {
    si (t[0]==3)
        afficheligne(3,t);
    sinon
        res.empile(3);
}


//Nous créons un tableau de 10 éléments.
tableau toto(10);

//Deux variables qui seront transmises aux tâches.
table r;
chaine s;

//Initialisation
toto[0]=2;
//On place ces tâches en attente avec deux paramètres r et s
toto.attentes(f1,f2,f3,r,s);

//Déclenchement des tâches suivi de leur réactivation
toto.déclenche(vrai);

//On change la valeur et on redéclenche l'ensemble, mais en laissant les tâches se
terminer.
toto[0]=3;
toto.déclenche(faux);
```

Type liste

Une liste ne se différencie d'une table que par la façon plus efficace d'y ranger les éléments par la tête ou la queue. Une liste permet d'implémenter des gestions où les extrémités sont plus importantes dans la fonctionnement que les éléments intermédiaires.

- Méthodes

1. applique(a,b,c): applique sur chaque élément une fonction, ou applique toutes les fonctions dans le conteneur
2. dépileenqueue(): retire le dernier élément de la liste
3. dépileentête(): retire le premier élément de la liste.
4. dernier(): renvoie le dernier élément de la table
5. empileenqueue(a): rajoute en queue de la liste
6. empileentête(a): ajoute en tête de la liste
7. insère(i,x): insère l'élément x à la position i
8. renverse(): renverse l'ordre des éléments dans la liste
9. joins(chaine sep): concatène chaque élément dans la liste en une chaine où chaque élément est séparé des autres par sep
10. premier(): renvoie le premier élément de la liste
11. produit(): Multiplie chaque élément avec les autres
12. raz(): nettoie la liste
13. somme(): Somme chaque élément
14. teste(i): teste si i est une position valide dans la liste

- Initialisation

Une liste peut être initialisée avec une structure entre "[]".

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

Opérateurs

x dans vliste: renvoie vrai ou une liste d'indexes, selon la variable de réception. Si la liste contient des chaines, alors la comparaison se fait sur une égalité stricte entre les chaines. Le système ne descend pas récursivement pour effectuer un « dans » sur les chaines présentes dans le conteneur. Pour obtenir ce comportement, il faut utiliser une fonction de comparaison (voir plus bas).

pour (s dans vliste) {…}: itère parmi toutes les valeurs. A chaque itération s contient une valeur de vliste.

+,*,-,/ etc..: ajoute etc.. une valeur à chaque élément d'une liste ou ajoute chaque élément d'une liste à une autre.

&,|: intersection ou union de deux listes

Comme nombre ou comme décimal

Renvoie la taille de la liste

Comme chaine

Renvoie une structure, où chaque élément est séparé des autres avec une virgule.

Indexes

On peut utiliser des indexes avec les listes, mais ceux-ci sont particulièrement inefficaces et devraient être évités.

Exemple

```
liste vliste=[1,2,3,4,5];

vliste.empileentête(10);
vliste.empileenqueue(20); //affichage: [10,1,2,3,4,5,20]
vliste.dépileentête();//affichage: [1,2,3,4,5,20]

table v=vliste; //transforme une liste en une table
```

Type dictionnaire (type dicotrié, dicopremier)

Un dictionnaire est un conteneur dans lequel les données sont sauvegardées via une clef. Cette clef est automatiquement transformée en chaine. La valeur peut être n'importe quel objet KIFF. Le type *dicotrié* fonctionne de la même façon, mais les clefs sont sauvegardées de façon ordonnées.

- Méthodes

1. amorce(table premiers) : Uniquement pour les *dicopremier.* Fournit la liste des nombres dont le système va se servir pour construire les différents niveaux de table de hachage.
2. applique(a,b,c): applique toutes les fonctions dans le conteneur.
3. applique(fonction,a,b,c): applique sur chaque élément une fonction. Attention, dans ce cas la fonction appelée doit avoir au moins deux paramètres, l'un contiendra la clef, le suivant la valeur, à chaque appel.
4. clefs(): renvoies les clefs du dictionnaire comme table
5. dépile(chaine clef): retire l'élément correspondant à clef
6. produit(): Multiplie chaque élément avec les autres
7. raz(): nettoie le dictionnaire
8. somme(): Somme chaque élément
9. teste(i): teste si i est une clef valide dans le dictionnaire
10. valeurs(): renvoie les valeurs du dictionnaire comme une table

- Initialisation

Un dictionnaire peut être initialisé avec une description telle que: {"k1":v1,"k2":v2...}

dictionnaire toto= {"a":1,"b":2};

- Opérateur

x dans adictionnaire: renvoie vrai ou une liste de indexes, selon la variable de réception. Si le dictionnaire contient des chaines, alors la

comparaison se fait sur une égalité stricte entre les chaines. Le système ne descend pas récursivement pour effectuer un « dans » sur les chaines présentes dans le conteneur. Pour obtenir ce comportement, il faut utiliser une fonction de comparaison (voir plus haut).

Important:

x est testé par rapport aux valeurs du dictionnaire et non les clefs. Il faut utiliser « teste » pour effectuer une telle vérification.

pour (s dans adictionnaire) {…}: itère parmi toutes les clefs. A chaque itération “s” contient une clef du dictionnaire.

+,*,-,/ etc..: ajoute etc.. une valeur à chaque élément d’un dictionnaire ou ajoute chaque élément d’un dictionnaire à un autre selon ses clefs

&,|: intersection ou union de deux dictionnaires selon les clefs.

Indexes

dictionnaire[clé]: renvoie l’élément dont *clé* est la clef. Si *clé* n’est pas une clef du dictionnaire, alors une exception est levée. Si en revanche, on place en début de code, un appel à la fonction *erreursurclef(faux),* KIFF renverra la valeur particulière *vide* à la place. *Par défaut, cette contrainte est à faux, initialement.*

Comme nombre ou comme décimal

Renvoie la taille du dictionnaire

Comme chaine

Renvoie une chaine qui correspond à la structure d’initialisation.

Exemple

```
dictionnaire vdictionnaire;

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

afficheligne(vdictionnaire);                    //affichage: {'10':27,'toto':1}
```

Tester les clefs

Il existe différentes méthodes pour vérifier l’existence d’une clef dans un dictionnaire. La première méthode est d’utiliser « teste » qui renverra vrai ou faux selon que la clef est présente ou absente. En

revanche, par défaut le test d'une clef sur un dictionnaire, si l'on utilise la représentation [] remonte une exception lorsque la valeur est absente.

dictionnaire d={1 :2, 3 :4} ;

d[5] renvoie une exception.

Il est possible d'éviter de tester systématiquement une exception, en désactivant l'erreur sur clef, via la fonction : *erreursurclef(faux),* placé en tout début de code. Lorsque cette fonction a été appelée (valable dès lors pour l'ensemble du programme), un test tel que d[5] renverra la valeur par défaut : *vide*. Il ne faut pas confondre cette valeur avec *nulle,* qui peut parfaitement être présent dans un dictionnaire comme valeur.

Conteneurs spécialisés

tableoctets, tablenombres, tabledécimaux, tablechaines, tableuchaines

Pour des raisons d'efficacité, KIFF fournit une série de conteneurs spécialisés n'acceptant qu'un seul type de valeur. Ces conteneurs contiennent un élément du type défini par leur propre nom.

tableoctets, tablenombres, *tabledécimaux*, *tablechaines, tableuchaines* sont des tables respectivement d'octets, de nombre, de décimaux, de chaines ou de chaines unicodes.

Remarque : *tableoctets* convient particulièrement à la lecture binaire des fichiers.

Dictionnaires spécialisés : dico…

Ces dictionnaires offrent toutes les combinaisons clef/valeur possibles, ajouté sous la forme de deux lettres à la fin du nom : c(haine), n(ombre), d(écimal), u(nicode).

- *dicoc, dicon*, *dicod, dicou* sont des dictionnaires dont les valeurs peuvent être n'importe quel objet, mais dont les clefs sont respectivement des chaines, des nombres, des décimaux ou des chaines unicodes. Le type *dictionnaire* est équivalent à *dicoc* par soucis de cohérence.

- *dicocn, dicocd, dicocc* sont des dictionnaires dont les valeurs sont respectivement des nombres, des décimaux ou des chaines, mais dont les clefs sont des *chaines*.

- *diconn, dicond, diconc, diconu* sont des dictionnaires dont les valeurs sont respectivement des nombres, des décimaux, des chaines ou des chaines unicodes, mais dont les clefs sont des *nombres*.

- *dicodn, dicodd, dicodc, dicodu* sont des dictionnaires dont les valeurs sont respectivement des nombres, des décimaux, des chaines ou des chaines unicodes, mais dont les clefs sont des *décimaux*.

- *dicoun, dicoud, dicouu* sont des dictionnaires dont les valeurs sont respectivement des nombres, des décimaux ou des

chaines unicodes, mais dont les clefs sont des *chaines unicodes*. Remarquons que *dicocu* n'existe pas. On suppose qu'une utilisation de chaines unicodes en clef impliquent que les valeurs en soient aussi.

Dictionnaires tries: dicotrié…

L'ensemble des dictionnaires ci-dessus trouve sa correspondence avec les dictionnaires triés suivants.

- *dicotriéc, dicotrién*, *dicotriéd, dicotriéu* sont des dictionnaires triés dont les valeurs peuvent être n'importe quel objet, mais dont les clefs sont respectivement des chaines, des nombres, des décimaux ou des chaines unicodes. Le type *dictionnaire* est équivalent à *dicotriéc* par soucis de cohérence.

- *dicotriécn, dicotriécd, dicotriécc* sont des dictionnaires triés dont les valeurs sont respectivement des nombres, des décimaux ou des chaines, mais dont les clefs sont des *chaines*.

- *dicotriénn, dicotriénd, dicotriénc, dicotriénu* sont des dictionnaires triés dont les valeurs sont respectivement des nombres, des décimaux, des chaines ou des chaines unicodes, mais dont les clefs sont des *nombres*.

- *dicotriédn, dicotriédd, dicotriédc, dicotriédu* sont des dictionnaires triés dont les valeurs sont respectivement des nombres, des décimaux, des chaines ou des chaines unicodes, mais dont les clefs sont des *décimaux*.

- *dicotriéun, dicotriéud, dicotriéuu* sont des dictionnaires triés dont les valeurs sont respectivement des nombres, des décimaux ou des chaines unicodes, mais dont les clefs sont des *chaines unicodes*. Remarquons que *dicotriécu* n'existe pas. On suppose qu'une utilisation de chaines unicodes en clef impliquent que les valeurs en soient aussi.

Note : Bon nombre de méthodes renvoit ou manipule des éléments de ce type.

tuple

Un tuple est conteneur dont la taille est fixe et dont les éléments ne sont pas modifiables.

Un tuple est représenté par une structure de type : (e1,e2…,en). Une particularité cependant de la déclaration des tuples est l'utilisation d'une écriture particulière pour les tuples de taille 1 : (e1,).

Exemple:

```
tuple t1=(10,);
tuple t3=(10,20,30);
```

tableau

Le dernier type, "tableau", est un conteneur dont la taille est fixée à la création.

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

Ce conteneur est très rapide, il est basé sur les tables telles qu'elles sont définies dans le langage C. Les éléments ne peuvent être atteint qu'à travers des indexes numériques.
Si la taille initiale définie à la création n'est pas suffisante, ou trop grande, il est possible de « retailler » ce tableau pour assurer un stockage optimal. On utilise à cette fin, la méthode « retaille » qui s'utilise de la façon suivante :

```
tableau test(10);
afficheligne(test.taille()); //10
test.retaille(20);
afficheligne(test.taille()); //la taille est maintenant de 20
```

Cette méthode permet non seulement de redimmensionner la taille du tableau, mais elle le fait tout en assurant que les éléments stockés dans le tableau soient bien enregistrés à leur place. Attention, si vous dimininuez la taille du tableau, les éléments placés au-delà de la nouvelle dimension seront perdus.

Important : ce type n'est pas protégé en lecture/écriture au sein d'une tâche. En cas d'utilisation au sein de tâches, si vous pouvez vous assurer qu'aucune collision (*lecture et écriture simultanées d'un même élément*) n'aura lieu pendant l'exécution, ce type de conteneur peut s'avérer très efficace, en réduisant notamment le nombre de verrous internes. Dans le cas opposé, il est fortement conseillé d'utiliser des verrous autour des manipulations des éléments d'un tableau pour éviter les plantages du moteur lui-même, en particulier lors d'une opération de redimentionnement.

attentes,déclenche

Ces deux méthodes ne sont accessibles qu'à travers le type *tableau*, justement du fait de l'absence de sémaphores internes pour ce type, ce qui permet d'effectuer une exécution simultanée des tâches listées dans *attentes*.

Type sac, sacchaines, sacnombres, sacdécimaux

On utilise les « sacs » pour conserver des ensembles d'éléments enregistrés dans un ordre « lâche ». Comme pour les autres conteneurs, il existe un « sac » générique et des sacs spécialisés.

Méthodes

1. dépile(val): retire du sac l'élément dont la valeur est *val.*
2. empile(val): rajoute *val* dans le sac
3. joins(chaine sep): concatène chaque élément dans le sac en une chaine ou chaque élément est séparé des autres par sep
4. octets(): Renvoie la chaine correspondant à une table d'octets. Chaque valeur doit être comprise entre 0..255
5. produit(): Multiplie chaque élément avec les autres
6. raz(): nettoie le sac
7. somme(): Additionne chaque élément avec les autres
8. teste(val): teste si *val* existe dans le sac.

Initialisation

On peut initialiser un sac avec une table.

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

Opérateurs

x dans *vsac*: renvoie vrai si *x* est dans *vsac*

pour (*s* dans *vsac*) {…}: itère parmi les valeurs. A chaque itération "s" contient une valeur de *vsac*.

+,*,-,/ etc..: ajoute (etc…) une valeur à chaque élément d'un sac ou ajoute chaque valeur d'un sac à un autre.

&,|: intersection ou union de deux sacs.

Comme nombre ou comme décimal

Il renvoie la taille de la table

Comme chaine

Il renvoie une structure, où chaque élément est séparé des autres par une virgule.

Indexes

Dans le cas d'un sac, les indexes sont des valeurs et non des indexes.

Exemple

```
sac vsac=[1,2,3,4,5];

vsac.empile(10);
vsac.empile(20);
vsac.dépile(10); //retire la valeur 10 du sac

table t=vsac; //conversion d'un sac en table...
```

Type grammaire

Ce type permet aux programmeurs de décrire facilement des chaines de caractères structurées.
Si vous devez, par exemple, détecter dans un texte des sous-chaines qui comprennent un mélange de caractères, de ponctuations ou de chiffre, une grammaire vous simplifiera la vie.

Méthodes

Ce type n'offre que deux méthodes:

1. **charge(règle):** *Vous pouvez charger vos règles sous la forme d'une chaine ou d'un vecteur de chaines, où chaque élément est une règle.*

2. **applique(chaine|table):** *Vous pouvez aussi bien appliquer cette grammaire à une chaine qu'à une table de chaines.*

Note: *L'opérateur "dans" utilise avec une grammaire, permet de détecter si une chaine est ou non reconnue par celle-ci.*

Règles

Vous pouvez fournir vos règles sous la forme d'une seule grosse chaine de caractères ou d'une table où chaque élément est une règle.

Format

Le format est le suivant:

tête:= (~) élément [,;] élément .

où élément est:

une chaine	=	soit "a" ou 'a'
?	=	n'importe quel caractère
%a	=	un caractère alphabétique
%C	=	une majuscule
%c	=	une minuscule
%d	=	un chiffre
%s	=	un espace
%p	=	une ponctuation
0,1,2..9	=	pour introduire le code d'un caractère (uniquement composé de chiffres)
$string	=	une chaine (peut aussi s'écrire "string")

tête	=	un appel à une autre règle
%?	=	le caractère ?
%%	=	le caractère %

- Négation: *Tous les éléments peuvent être niés sauf un appel de règle.*
- Disjonction: *Le ";" permet d'exprimer la disjunction entre deux éléments, sinon utilisez la ",".*
- Convention de Kleene: *Vous pouvez utiliser la « * » ou la « + » pour obtenir des boucles sur certains éléments.*
- Optionalité: *Utilisez les parentheses pour rendre un élément optionel.*
- Toutes les règles doivent se terminer avec un "."
- Quand le nom de la tête d'une règle commence avec un « _ », cela signifie que le nom de cette règle n'apparaitra pas dans le résultat final. Celle-ci est quand même appliquée.

Autres cas:

?_	=	n'importe quel caractère, mais non gardé
%a_	=	un caractère alphabétique, mais non gardé
%C_	=	une majuscule, mais non gardé
%c_	=	une minuscule, mais non gardé
%d_	=	un chiffre, mais non gardé
%s_	=	un espace, mais non gardé
%p_	=	une ponctuation, mais non gardé
label_	=	un appel à une règle, sans garder le résultat

Le rajout d'un "_" à la fin de chacune de ces options, conserve la nature de l'option, mais retire du résultat final le caractère détecté par celle-ci.

Exemple

```
//Cette grammaire reconnait un nombre et un mot
chaine r=@"

bloc  := mot;nombre.
mot := %a+.
nombre := %d+.

"@;

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

//we apply it to the string the
dictionnaire m=g.applique("the"); //renvoie: {'bloc':[{'mot':['the']}]}
```

```
m=g.applique("123"); //renvoie: {'bloc':[{'nombre':['123']}]}
```

Cependant, si nous appliquons cette grammaire à la chaine: “Test 123”, celle-ci échouera.
Nous avons besoin d’enrichir notre grammaire avec 2 choses:

a) Tout d’abord, elle doit prendre en compte les espaces.
b) Elle doit aussi être capable de boucler sur chaque élément

```
chaine r=@"

base := bloc+.
bloc  := mot;nombre;%s.
mot := %a+.
nombre := %d+.

"@;
```

Nous avons rajouté le %s pour prendre en compte les espaces, puis nous avons introduit le règle “base” pour effectuer une boucle sur les blocs.

Si nous appliquons notre grammaire à: “Test 123”, le système renvoie:

{'base':[{'bloc':[{'mot':['Test']}]},{'bloc':[' ']},{'bloc':[{'nombre':['123']}]}]}

Cependant, cette structure peut s’avérer difficile à lire. Nous pouvons alors utiliser l’opérateur « _ » pour retirer les éléments redondants, tel que « bloc » :

```
chaine r=@"

base := _bloc+.
_bloc  := mot;nombre;%s.
mot := %a+.
nombre := %d+.

"@;
```

Désormais _bloc est une tête cachée de cette grammaire, ce qui nous permet de simplifier notre résultat en:

{'base':[{'mot':['Test']},' ',{'nombre':['123']}]}

Nous pourrions aussi décider de reconnaitre non seulement les chiffres mais aussi les nombres en toute lettre. Remarquons qu’alors nous plaçons *nombre* en tête de la règle pour éviter une ambiguité avec la reconnaissance d’un mot.

```
chaine r=@"

base := _bloc+.
_bloc  := nombre;mot;%s.
mot := %a+.
nombre := %d+;$milliard;$million;$mille.

"@;
```

Appliquons cette grammaire à: “Test millions de veaux”, nous obtenons:

```
{'base':[{'mot':['Test']},' ',{'nombre':['millions']},' ',{'mot':['de']},' ',{'mot':[veaux']}]}
```

Pour reconnaitre des structures plus compliquées telles que des codes, nous pourrions implanter la grammaire suivante:

```
chaine r=@"

base := _bloc+.
_bloc  := code;mot;nombre;%s.
mot := %a+.
nombre := %d+.
code := %C,%d+,%c.

"@;
```

Si nous appliquons cette grammaire à: “Test 123 T234e”, nous obtenons:

```
{'base':[{'mot':['Test']},' ',{'nombre':['123']},' ',{'code':['T234e']}]}
```

Sous-grammaires

Les sous-grammaires sont introduites au sein d’une règle avec des […]. Entre ces crochets, on peut alors définir une liste disjointe d’expressions régulières qui s’appliqueront au niveau des caractères. Ces expressions sont très utiles lorsque l’on travaille avec une table de chaines. Il devient alors possible d’établir une correspondance entre la chaine issue de la table et une description de cette chaine. Chaque expression doit être séparée de la suivate avec un « | ». Attention, on ne peut pas appeler d’autres règles depuis une telle expression, par conséquent une expression de type : toto, sera équivalente à $toto.

Example:

```
chaine dico=@"

test := %a, mot,%a.

wrd := [%C,("-"),%c+|test|être|chien|chat].

"@;
```

```
grammaire g(dico);

uchaine s="Le C-at boit";

tableuchaines v=s.segmente();

table res=g.applique(v);
```

Table vs. Dictionnaire

Si la variable de réception est une table au lieu d'un dictionnaire, le résultat rendu est alors très différent. En particulier, on place en tête de chaque table, le nom de la règle qui a produit la structure.

Si l'on applique notre grammaire toujours sur la même entrée mais avec une table comme réception, nous obtenons :

['base',['mot','Test'],' ',['nombre','123'],' ',['code','T234e']]

La structure en entrée est une chaine ou une table

Si l'entrée est une chaine, nous concaténons chaque caractère détecté au résultat. Si en revanche, il s'agit d'une table, dans ce cas, nous gardons le résultat sous la forme d'une table de caractères.

Example

```
//Cette grammaire reconnait un nombre ou une chaine
chaine r=@"

base := _bloc+.
_bloc  := code;mot;nombre;%s.
mot := %a+.
nombre := %d+.
code := %C,%d+,%c.

"@;

//Nous la chargeons
grammaire g(r);

//nous explosons notre chaine en une table de caractères
chaine c="Test 123 T234e";
table te=c.éclate("");

//we apply the grammar to the character vector
table  t=g.applique(te);
afficheligne(t);
```

Ce qui nous donne:

['base',['mot','T','e','s','t'],' ',['nombre','1','2','3'],' ',['code','T','2','3','4','e']]

Fonction

On peut enfin associer une fonction à une grammaire, dont la signature est la suivante :

fonction appelgrm(chaine tête, omni structure,nombre pos).

Cette fonction est appelée chaque fois qu'une structure a été déterminée par une règle. Si cette fonction renvoie *faux*, alors l'application de la règle échoue aussi. *pos* correspond à la dernière position atteinte dans le texte traité.

Exemple

```
chaine r=@"

base := _bloc+.
_bloc  := code;mot;nombre;%s.
mot := %a+.
nombre := %d+.
code := %C,%d+,%c.

"@;

//La fonction appelée
fonction appelgrm(chaine tête,omni t,nombre pos) {
    afficheligne(tête,t,pos);
    renvoie(vrai);
}

grammaire g(r) avec appelgrm;

chaine s="Test 123 T234e";
dictionnaire m=g.applique(s);
afficheligne(m);
```

Résultat:

```
mot ['Test']
_bloc [{'mot':['Test']}]
_bloc [' ']
nombre ['123']
_bloc [{'nombre':['123']}]
_bloc [' ']
code ['T234e']
_bloc [{'code':['T234e']}]

{'base':[{'mot':['Test']},' ',{'nombre':['123']},' ',{'code':['T234e']}]}
```

Modification de la structure

Il est aussi possible d'apporter des modifications à la structure dans une telle fonction. A vos propres risques…

```
fonction appelgrm(chaine tête,omni t,nombre pos) {
    si (tête=="mot") {
        afficheligne(tête,t);
        t[0]+="_mot";
    }
    renvoie(vrai);
}
```

Dans ce cas nous obtenons:

```
mot ['Test']

{'base':[{'mot':['Test_mot']},' ',{'nombre':['123']},' ',{'code':['T234e']}]}
```

Depuis l'intérieur d'une règle

Une function peut aussi être appelé depuis l'intérieur d'une règle. La signature de la fonction est la suivante.

fonction appelregle(omni structure,nombre pos).

```
//Cette fonction est appelée depuis la règle code
//Si elle renvoie faux, la règle échoue
fonction appelcode(omni t,nombre pos) {
    afficheligne(t,pos);
    renvoie(vrai);
}

chaine r=@"

base := _bloc+.
_bloc  := code;mot;nombre;%s.
mot := %a+.
nombre := %d+.
code := %C,%d+,%c,appelcode.

"@;

grammaire g(r);

chaine s="Test 123 T234e";
dictionnaire m=g.applique(s);
afficheligne(m);
```

Exemple : Analyser de l'HTML

```
//évalue est une méthode de base pour les chaines qui permet de transformer toutes les formes d'entités HTML en caractère UTF8
fonction evalue(omni s,nombre p) {
    s[1]=s[1].évalue();
    renvoie(vrai);
}

//La grammaire est fournie sous la forme d'une chaine
chaine htmlgrm=@"

html := _objet+.
_objet := balise;%s_;texte.
balise := "<",?+,">".
texte := _caracteres,evalue.
_caracteres := ~"<"+.

"@;

//On compile la grammaire
grammaire ghtml(htmlgrm);

//On l'applique ensuite sur une chaine de caractère contenant de l'HTML
tablechaines rgram=ghtml.applique(texte_html);
```

Type arbre

Ce conteneur est utilisé pour gérer les données sous la forme d'un arbre.

Méthodes

1. derniernoeud()|derniernoeud(arbre a): renvoie le dernier noeud, ou compare a au dernier noeud.
2. élague(): Détruit le sous-arbre de l'arbre global.
3. fils()|fils(arbre a): renvoie le noeud fils, ou l'ajoute comme noeud fils.
4. frère()|frère(arbre a): renvoie le noeud frère, ou l'ajoute comme noeud frère.
5. isole(): Extrait le noeud courant de son arbre.
6. père()|père(arbre a): renvoie le noeud père, ou compare a au noeud père.
7. précédent()|précédent(arbre a): renvoie le noeud précédent, ou l'ajoute comme noeud precedent.
8. profondeur(): Renvoie la profondeur du noeud dans le arbre.

Opérateur

x dans arbre: renvoie vrai ou une liste de noeuds d'arbre, selon la variable de réception.

pour (s dans arbre) {…}: itère parmi toutes les clefs.

Comme chaine

Renvoie la valeur du nœud comme chaine

Comme nombre ou décimal

Renvoie la valeur du nœud comme un nombre ou un décimal

Exemple

```
//Parcours récursif
fonction treeaffichage(arbre t) {
        si (t==nulle)
                renvoie;
        affiche(t," ");
        si (t.fils()!=nulle) {
                affiche("("); //sous-noeuds sont affichagés entre “(…)”
                treeaffichage(t.fils());
                affiche(")");

        }
        treeaffichage(t.frère());
}

//nous créons cinq noeuds, avec des valeurs numériques
arbre test1(1);
arbre test2(2);
arbre test3(3);
arbre test4(4);
arbre test5(5);

test1.fils(test2);
test1. fils(test3);
test2. fils(test4);
test4.frère(test5);

treeaffichage(test1); //nous affichons: 1 (2 (4 5 )3 )

//Nous modifions la valeur de test5
test5=[100,200];

treeaffichage(test1); //nous affichons: 1 (2 (4 [100,200] )3 )

//nous retirons test4
test4.élague();
treeaffichage(test1); //nous affichons: 1 (2 ([100,200] )3 )


//nous utitisons nos valeurs dans une addition
nombre cpt=test1+test2+test3;
afficheligne(cpt); //nous affichons 6

omni u; //Comme nous ne savons rien sur les valeurs nous utitisons un type omni
pour (u dans test1)
  affiche(u,"[",u.prfondeur(),"] ");//affichage: 1[0] 2[1] [100,200][2] 3[1]
```

Type itérateur, ritérateur

Les itérateurs sont utilisés pour parcourir des objets de type: chaine, table, dictionnaire.

ritérateur est l'itérateur inverse, qui est utilisé pour parcourir depuis la fin de la collection.

- Méthodes

1. applique(a,b,c): applique une fonction
2. clef(): renvoie la clef de l'élément courant
3. commence(): initialise l'itérateur avec le début de la collection
4. fin(): renvoie vrai quand la fin de la collection est atteinte
5. nfin(): renvoie vrai tant que la fin de la collection n'a pas été atteinte (~fin())
6. metvaleur(valeur) : change la valeur courante pointée par l'itérateur
7. suivant(): élément suivant dans la collection
8. valeur(): renvoie la valeur de l'élément courant
9. valeurtype(): renvoie le type de la valeur de l'élément courant

- Initialisation

Un itérateur est initialisé avec une simple affectation.

- Exemple

```
table v=[1,2,3,4,5];
itérateur it=v;
pour (it.commence() tantque it.nfin() faire it.suivant())
        affiche(it.valeur(),",");
```

Exécute
1,2,3,4,5,

Type date

Ce type est utilisé pour gérer les dates.

Méthodes

1. date(): renvoie la date comme chaine
2. année(): renvoie l'année comme un nombre
3. mois(): renvoie le mois comme un nombre
4. jour(): renvoie le jour comme un nombre
5. heure(): renvoie l'heure comme un nombre
6. metdate(année,mois,jour,heure,min,sec): initialise une variable de temps

Opérateurs

+,-: Les dates peuvent être ajoutées ou soustraites.

Comme chaine

Renvoie la date comme chaine

Comme nombre ou décimal

Renvoie le nombre de secondes écoulées depuis 00:00 heure, Jan 1, 1970 UTC

Exemple

```
date mytemps;

affiche(mytemps);          // affichage: 2010/07/08 15:19:22
```

Type chrono

Ce type est utilisé pour mesurer des durées très courtes, de l'ordre de la milliseconde.

- Méthodes

1. raz (): réinitialise une variable chrono

- Opérateurs

+,-: Les chronos peuvent être ajoutés ou soustraits

- Comme chaine

Renvoie le temps en ms

- Comme nombre ou décimal

Renvoie le temps en ms

- Exemple

```
chrono letemps;

affiche(letemps);
```

Type fichier

Ce type est utilisé pour gérer des fichiers en *entrée/sortie*.

- Méthodes

1. cherche(chaine c,bool sanscasse): Cherche une chaine dans un fichier et renvoie toutes ses positions.
2. dit(): renvoie la position du curseur courant dans le fichier
1. écrit(chaine s1,chaine s2,): écrit les chaines dans le fichier
3. écritbin(s1, s2,): écrit les codes de caractères dans le fichier. Si le paramètre est un conteneur, alors chaque élément est considéré comme un caractère. Il est préférable d'utiliser un tableoctets à cette fin.
4. findefichier(): renvoie vrai quand la fin de fichier est atteinte
5. litln(): lit une ligne depuis un fichier
6. lit(): lit tout le document dans une variable. Si la variable de réception est un tablechaines, alors le fichier est découpé le long des retours chariots et chaque chaine est sauvegardée dans le conteneur. Si le conteneur est un tableoctets ou un tablenombres, chaque octet du fichier est sauvegardé séparément dans le conteneur.
7. lit(nombre nb) : Même chose que lit, mais met une limite au nombre de caractères extraits du fichier.
8. litun(): lecture d'un caractère depuis le fichier
9. ouvreenajout(chaine fichiername): ouvre un fichier en ajout
10. ouvreenécriture(chaine fichiername): ouvre un fichier en écriture
11. ouvreenlecture(chaine fichiername): ouvre un fichier en lecture
12. positionne(nombre p): positionne le curseur du fichier à p
13. remet(nb): remet nb caractères dans le flux
14. fichier f(chaine fichiernom, chaine modeouverture): ouvre un fichier selon les modes de lectures suivants:

a. “a”: ajout
b. “l”: lire
c. “e”: écrire
d. “e+”: ajout

Opérateur

x dans fichier: si x est une chaine, alors Il reçoit le fichier ligne à ligne, si c’est une table, Il empile les lignes dedans. Si x est un nombre ou un décimal, Il le fichier caractère par caractère.

Exemple

```
fichier f;
f.ouvreenlecture(chemin);
chaine s;
table mots;
chaine w;
pour (s dans f) {//Utilise l’opérateur dans
        s=s.rogne();
        s.éclate(" ",mots);
        pour (w dans mots)
                affiche("mot:",w,finl);
}
f.fin();
```

Entrée standard: entréestandard (stdin)

KIFF fournit la variable entréestandard pour gérer l’entrée standard (*ou standard input : stdin en anglais*). Cette variable peut être utile pour traiter un fichier lu à travers un « pipe ».

Exemple

```
nombre s;
nombre i=1;
pour (s dans entréestandard) {
  afficheligne(i,s);
  i++;
}
```

Si vous exécutez le fichier ci-dessus, vous afficherez la ligne code de caractère par code de caractère.

echo “Le chien est content” | KIFF stdin.kiff.

Type appel

Cet objet est utilisé pour enregister un pointeur sur une fonction, qui peut alors être exécutée via une variable de ce type. Une variable de type *appel* est initialisée avec le nom de la fonction elle-même.

Note :

Une variable de type *appel* peut remplacer le nom de la fonction après un opérateur *avec.* Dans ce cas, la fonction de rappel peut être modifiée de façon dynamique.

Exemple

```
fonction affichage(nombre e)  {
        affiche("AFFICHAGE:",e,"\n");
        e+=10;
        renvoie(e);
}

appel myfunc;
myfunc=affichage;         // On lui donne le nom de la fonction
nombre i=myfunc(100);   // affichage: AFFICHAGE:TEST
afficheligne("I=",i);                  //affichage: I=110
```

Type xmldoc

Ce type est utilisé pour gérer des documents XML. Il est possible d'associer une fonction de rappel avec une variable xmldoc de façon à pouvoir accéder à chaque nœud xml à la volée pendant la lecture du fichier.

Méthodes

1. charge(chaine nomfichier): charge un fichier xml
2. ferme(): Ferme le fichier XML
3. crée(chaine topnoeud): Crée un fichier XML avec topnoeud comme noeud racine. Si topnoeud est une structure XML complète, alors il crée le fichier sur cette base.
4. àlafermeture(fonction,objet): Fonction de rappel pour les balises fermantes
5. analyse(chaine tampon): analyse une structure xml fournie sous la forme d'une chaine de caractères.
6. xpath(chaine myxpath): Renvoie une table de noeuds xml correspondant au chemin xpath myxpath. Les chemins *xpath* obéissent à des règles précises, qui ne sont pas décrites ici, mais dont la description peut être trouvée sur les sites spécialisées.
7. sauvegarde(chaine nomfichier): sauvegarde
8. noeud(): Retourne le noeud racine
9. chainexml(): Renvoie un document XML sous la forme d'une chaine.
10. sérialise(objet): Sérialise un objet KIFF sous la forme d'un document XML
11. sérialisechaine(objet): Sérialise un objet KIFF sous la forme d'un document XML et renvoie la chaine correspondante

Fonctions de rappel

Les fonctions de rappel doivent avoir la signature suivante :

fonction xmlnoeud(xml n, objet);

Selon la déclaration suivante:

xmldoc mydoc(obj) avec xmlnoeud;

Type xml

Le type xml offre les méthodes nécessaires pour gérer un nœud XML.

Important

Ce type est implémenté comme encapsulateur d'un pointeur *xmlNoeudPtr* de la bibliothèque libxml2 library (voir http://xmlsoft.org/), d'où la méthode *nouveau* qui est nécessaire pour obtenir un nouveau nœud XML.

Méthodes

1. chainexml(): Retourne l'arbre complet depuis le noeud courant sous la forme d'une chaine de caractères.
2. contenu(): Retourne le contenu textuel du noeud xml
3. délie(): Retire un noeud de la structure arborée
4. détruit(): Détruit la représentation interne du noeud XML
5. enfant(): noeud xml enfant
6. id(): Renvoie l'identifieur du noeud (uniquement disponible avec les fonctions de rappel)
7. ligne(): Renvoie la ligne où se trouve le noeud
8. namespace(): Retourne la table de namespace du noeud xml
9. nom(): nom du noeud
10. nouveau(chaine name): Crée un nouveau noeud XML
11. parent(): noeud xml parent
12. précédent(): noeud xml précédent
13. propriétés(): Retourne les propriétés du noeud xml sous la forme d'un dictionnaire dont les clefs sont les noms des attributs du nœud.
14. racine(): Retourne le noeud racine

15. suivant(): noeud XML suivant

16. xmltype(): Renvoie le type du noeud XML

- Comme chaine

Renvoie le nom du nœud XML

Exemple

```
fonction test(xml n, omni nn) {
    dicocc m=n.propriétés();
    afficheligne(n.nom(),m,n.contenu());
}


xmldoc doc avec test;

doc.charge("resxip.xml");

xml nd=doc.noeud();

afficheligne(nd);

tantque (nd!=nulle) {
    afficheligne(nd.contenu(),nd.namespace());
    nd=nd.enfant();
}


xmldoc nouveau;

nouveau.crée("TESTAGE");
xml nd=nouveau.noeud();

xml n("toto");
nd.enfant(n);

n.nouveau("titi");

n.contenu("Toto est content");
nd.enfant(n);

nouveau.sauvegarde("mynouveaufichier.xml");
```

Type kiff

Le type *kiff* est utilisé pour charger un programme KIFF dynamiquement, mais aussi pour gérer des sessions KIFF où le programme est fourni sous la forme de chaines de caractères.

Méthodes

1. kiff var(chaine kiffnomchemin): Crée et charge un programme KIFF
2. _chargeur: Une variable de type kiff, qui garde l'origine du programme appelant.
3. charge(chaine kiffnomchemin): charge un programme KIFF
4. compile(chaine kiffnomchemin): Compile un programme KIFF sous la forme d'une chaine. Renvoie un identifiant sur la première instruction à exécuter
5. exécute(nombre i): Exécute un programme depuis l'instruction i (renvoyé par compile)
6. exécutetâche(nombre i): Exécute en parallèle un programme compilé depuis l'identifiant renvoyé
7. exportées(): Renvoie la liste des méthodes exportées
8. fermedebug(): Fin du mode debug commencé avec fonctiondebug
9. finexécution(): Renvoie vrai si le programme s'est entièrement exécuté
10. fonctiondebug(fonction,objet): Initialise la fonction de debuggage qui sera appelé pendant l'exécution
11. kvariables(): Renvoie les variables actives sous la forme d'un dictionnaire
12. nom(): Renvoie le chemin du fichier KIFF
13. ouvre(): ouvre une session KIFF
14. nettoie(): ferme une session KIFF

Exécuter des fonctions présentes dans un fichier externe

Lorsque l'on charge un programme externe, ses méthodes, du moins celles qui ne sont pas déclarées *privée,* deviennent dès lors disponibles pour le programme appelant. On les appelle via la variable *kiff* déclarée à travers laquelle le programme a été chargé.

Exemple

Dans notre programme test.kiff, nous implémentons la fonction: Lecture

test.kiff

```
fonction Lecture (chaine s) {
        //nous rappelons une fonction issue du programme appelant.
        _chargeur.Fin("De 'call' avec love");
        renvoie(s+"_toto");
}
```

appel.kiff

Dans notre programme appelant, nous chargeons test.kiff, puis nous exécutons Lecture

```
kiff kf;
kf.charge('c:\test.kiff'); //nous chargeons le fichier implémentant Lecture
chaine s=kf.Lecture("xxx"); //nous pouvons exécuter Lecture dans notre programme local.

//nous implémentons une fonction locale, qui sera appelée depuis test à travers _chargeur…
fonction Fin(chaine s) {
        afficheligne("Nous revenons:",s);
}
```

Comme chaine

Renvoie le nom du fichier KIFF.

Comme Booléen

Renvoie vrai si un fichier a été chargé.

Lecture croisée

Si l'utitisateur charge le même programme KIFF dans plusieurs fichiers différents, les variables de ce programme ne seront déclarées qu'une seule fois.

Si prg1.KIFF charge prg2.KIFF et prg2.KIFF charge prg3.kiff, qui charge prg1.KIFF à son tour. Le second prg1.KIFF ne sera pas chargé et un pointeur sur l'espace mémoire occupé lors du premier

chargement sera renvoyé, autorisant ainsi prg2 et prg3 à partager les mêmes variables issues de prg1.

Il est possible d'initialiser dans un programme P1, une référence au programme P2 via une fonction déclarée dans P1, appelé avec comme paramètre : *ici*. *Ici* est un mot clef qui fait référence à l'instance courante d'un fichier chargé.

prg1.kiff

```
kiff kf('c:\prg2.kiff'); //nous chargeons un fichier implémentant une méthode met
kf.met(ici); //nous initialisons notre KIFF local dans prg2.kiff.
```

prg2.kiff

```
kiff appeleur;

//Cette fonction est appelée depuis prg2 et instancie appeleur avec
//une référence au chargeur. prg2 peut alors appeler des fonctions implémentées
dans //prg1.kiff

fonction met(kiff c) {
   appeleur=c;
}
```

_chargeur

Dans cet exemple, *appeleur* et *_chargeur* vont partager la même valeur.

privée

Si vous ne désirez pas que certaines fonctions soient accessibles depuis un programme externe, vous pouvez les déclarer comme *privée*.

Exemple

```
//nous implémentons une fonction, qui ne peut être appelé depuis l'extérieur
privée fonction Nepeutêtreappelée(chaine s) {…}
```

chargedans

KIFF fournit aussi la fonction *chargedans* qui fusionne dans l'espace courant le contenu d'un autre fichier. Cette fonction doit s'utiliser en début de programme et ne peut être appelée dynamiquement.

Exemple

```
chargedans('c:\fichiers\prgm.kiff'); //ce programme contient vccc, une table
afficheligne(vccc); //qui est désormais disponible dans le programme courant.
```

Session: ouvre, nettoie, compile, exécute

Une session permet de compiler et d'exécuter du code fourni sous la forme de chaines de caractères.

Exemple

Dans l'exemple qui suit nous ouvrons une session et compilons une simple ligne de code, que nous exécutons ensuite.

```
KIFF session;
session.ouvre();                // nous ouvrons une session
nombre premier=session.compile("nombre i=10;"); //nous compilons la ligne, qui renvoie la position de la première instruction en mémoire
session.compile("nombre j=10;");     //nous ajoutons d'autres lignes
sessions.compile("affiche ligne(i,j);"); //ainsi qu'un affiche ligne
session.exécute(premier);// nous exécutons en utitisant la valeur rendue par le premier compile : 10 10
nombre second=session.compile("j=i+10;"); //nous ajoutons d'autres lignes...
session.compile("affiche ligne(i,j);"); //que nous compilons

session.exécute(second);//nous exécutons de nouveau mais cette fois depuis second: 10 20

session.nettoie(); //fin de la session
```

Instructions particulières

KIFF fournit toutes les méthodes algorithmiques nécessaires à l'écriture de programmes riches et complexes.

si—sinonsi—sinon

si (expression) {}

sinonsi (expression) {}

…

sinon {}

Opérateurs Booléens : *ou, et*

Les expressions Booléennes sont au cœur de la majorité des tests. Dans KIFF, on peut utilise « et » et « ou » pour coordonner entre eux les différents éléments d'une telle expression.

si (a=="test" ou a=="autre") …

parmi (expression) (avec fonction) {…}

parmi permet de factoriser une série de test pour une variable:

```
parmi(expression) {
          v1 : {…
          }
          v2 : {…
          }
          défaut: {…        //défaut est un mot clef prédéfini
    }
}
```

v1,v2,..vn peuvent être une chaine, un nombre ou un décimal ou tout autre objet.
On peut aussi utiliser une fonction de comparaison qui permet d'intercepter la comparaison de la variable avec ses valeurs. Cette fonction doit renvoyer vrai ou faux.

```
fonction test(nombre i,nombre j) {
```

```
        si (j>=i)
            renvoie(vrai);
        renvoie(faux);
}
nombre s=10;
//La fonction test est utilisée pour effectuer les comparaisons.
switch (s) avec test {
        1:  afficheligne("1");
        2:  afficheligne("2");
        20:  afficheligne("20"); //choix
}
```

pour (expressions tantque test de continuation faire incréments)

Cet opérateur permet de mettre en place des boucles avec initialisation, test de continuation et itération en une seule structure.

pour est composée de trois parties :

1. Une initialisation
2. Un test de continuation
3. Un phase d'incrémentation

pour (expression tantque booléen faire suivant) {…}

Vous pouvez aussi utiliser : *continue* ou *arrête* pour contrôler le fonctionnement de la boucle.

Exemple

```
pour (i=0 tantque i<10 faire i+=1) affiche("I=",i,"\n");
```

Multiples initialisations et incréments

On peut rajouter plus d'une initialisation et plus d'un incrément, chacun devant être séparé des autres par une virgule.

Exemple

```
nombre i,j;

//Multiples initialisations et multiple incréments.
pour (i=10,j=100 tantque i>5 faire i--,j++)
        afficheligne(i,j);
```

- pour (var dans conteneur) {…}

Il s'agit d'un cas particulier de boucle : *pour,* utilisée pour itérer sur un conteneur, une chaine ou un fichier.

Exemple

```
//nous itérons dans un fichier
fichier f('myfichier.txt',"l");  //ouverture en lecture
chaine s;

pour (s dans f)
    afficheligne(s);
f.ferme() ;

//nous itérons dans une table de nombres...
table v=[1,2,3,4,5,6];
nombre i;

pour (i dans v)
    afficheligne(i);
```

tantque (booléen)

tantque attend une simple expression Booléenne.

tantque (booléen) {…}

On peut à tout moment rompre ou continuer au sein d'une boucle, grâce à *continue* ou *arrête*.

Exemple

```
nombre i=10;
tantque (i>0) {
        afficheligne("I=",i);
        i-=1;
}
```

faire {..} tantque (booléen) ;

Cette expression est similaire à *tantque* à la différence que la première itération est faite avant le test booléen.

```
nombre i=10;
faire {
        afficheligne("I=",i);
        i-=1;
}
tant que (i>0);
```

affiche,afficheligne,afficheerr,afficheligneerr

Ces instructions sont utilisées pour afficher des résultats à l'écran. Les versions en « err » envoient les informations à afficher sur « stderr» à l'écran. Les versions contenant « ligne » rajoutent deux particularités à l'affichage, tout d'abord chaque argument est séparé des suivants par un blanc, puis à la fin de l'affichage, un retour chariot est ajouté.

affichej,affichejligne,affichejerr,affichelignejerr

Ces instructions sont utilisées pour afficher des résultats à l'écran. Ces fonctions prennent en premier argument un conteneur et elles se servent des arguments suivants (au plus trois au total) pour effectuer un « joint » avant d'afficher le contenu à l'écran. Pour un conteneur de type « table », la fonction ne peut utiliser qu'un seul séparateur. Pour un conteneur de type « dictionnaire » on peut fournir deux séparateurs supplémentaires, l'un pour les clefs, l'autre pour les valeurs.
Si seul le conteneur est donné en argument, alors le séparateur par défaut un retour chariot.

Exemple

```
tablenombres v=[1..10];
affichej(v,"-");
```

Résultat: 1-2-3-4-5-6-7-8-9-10

```
dictionaire m={1:2,2:3,4:5,6:7};
affichej(m,"-",",");
```

Résultat: 1-2,2-3,4-5,6-7

redirigeio et réinstaureio

Ces deux fonctions servent à capturer les flux envoyés à *stderr* ou à *stdout*.

nombre redirigeio(chaine fichiersortie,bool err);
Cette fonction redirige vers le fichier soit le flux provenant de *stderr* (*err* est *vrai)* soit celui de *stdout (err* est *faux).* Il renvoie alors un identifiant nécessaire pour ramener le flux à la normale.

réinstaureio(nombre id, bool err);
Cette function ramène l'affichage à la normale. Il utilise à cette fin, l'identifiant renvoyé par *redirigeio.* Il ferme aussi le fichier à ce moment-là.

Exemple:

```
nombre o=redirigeio('C:\xip\test\test.txt',vrai);
affichelignеerr("Cette chaine est maintenant enregistrée dans: test.txt");
réinstaureio(o,true); //Retour à la normale.
```

suspend et pause

Ces deux fonctions sont utilisées pour suspendre une tâche ou la mettre en pause pendant un certain nombre de secondes.

- *pause* prend deux paramètres, l'un *décimal* pour définir le temps de pause, le second pour afficher ou non une petite animation. *Pause* ne suspend pas le fonctionnement d'une tâche.

- *suspend* est basé sur une implantation système qui suspend la fonctionnement d'une tâche. Il prend en paramètre un nombre dont la signification (seconde ou milliseconde) dépend de la plate-forme.

Exemple:

```
pause(0.1); //la tâche se met en pause pendant 10 ms
pause(2,vrai); //la tâche se met en pause pour 2s, avec une petite animation
suspend(1); //la tâche est suspendue pendant 1s (selon la plate-forme)
```

aléatoire()

KIFF fournit une fonction qui renvoie une valeur décimalle aléatoire, comprise entre 0 et 1.

Exemple:

```
décimal rd=aléatoire();
```

Touche frappée: touche()

touche() permet de saisir à la volée la frappe d'une touche sans afficher son contenu à l'écran.

Exemple

```
nombre c;
chaine message;
tantque (message!=".") {
        affiche(">");
```

```
        c=0;
        message="";
        tantque (c!=13 et c!=10) {
                c=touche();
                si (c!=13 et c!=10)
                        message+=c.car();
                affiche(c.car());
        }
        afficheligne();
        afficheligne("Fin:",message);
}
```

utilise(OS,bibliothèque)

utilise permet de charger des bibliothèques dynamiques compatible avec KIFF, pour rajouter de nouvelles fonctionnalités, telles que interface graphique, gestion de base de données etc.
Le premier argument est optionnel, il ne peut prendre que l'une des valeurs ci-dessous:

"WINDOWS" , "MACOS", "UNIX", "UNIX64".

Cette option permet de spécialiser les chargements de bibiliothèques par plate-formes.
bibliothèque peut être un simple nom, qui doit correspondre au nom d'une bibliothèque dynamique présente dans le répertoire dont le chemin est enregistré dans la variable d'environnement KIFFLIBS.
bibliothèque peut aussi être un chemin complet vers cette même bibliothèque, mais dans ce cas cet appel sera spécifique à une plate-forme donnée.

Convention de nommage des bibliothèques

- Sur Unix, les noms des bibliothèques sont généralement de la forme: lib*monnom*.so. Pour charger une telle bibliothèque, un simple appel: **utilise("monnom");** suffit.

- Sur Windows, les noms sont de la forme: *monom*.dll. Pour charger une telle bibliothèque, un simple appel: **utilise("monom")** suffit.

Il est généralement préférable d'écrire : utilise("monnom") de façon à permettre au code de s'exécuter de la même façon sur toutes les plate-formes. Cependant, vous pouvez aussi appeler directement la bibliothèque en fournissant son nom complet. L'utilisation des options d'OS permet alors de spécialiser ces appels par plate-formes.

```
utilise("WINDOWS", "kifflinear") ;
```

Exceptions : *tente, capte, lève*

Ces trois instructions sont utilisées pour la gestion des erreurs. *capte* peut être ou non associé avec une variable de chaine ou de nombre, qui sera automatiquement mise à zéro au moment de l'appel à *tente*.

```
chaine s;
tente {…
}
capte(s);
```

```
tente {
}
capte ;
```

```
tente {
}
capte {…}
```

Quand une erreur est détectée, celle-ci est automatiquement affectée à la variable dans *capte*.
On peut aussi décider d'exécuter dans *capte* une série d'instruction, par exemple pour afficher l'erreur.

- Lever une exception

1. lève(chaine s): lève une erreur avec le message s. Un message d'erreur doit comporter une chaine suivie d'un numéro entre parenthèses et du message d'erreur.

- Exemple:

```
lève("MON(201): Mon erreur");
```

Opérateur *dans*

Cet opérateur est assez complexe d'où la section spéciale qui lui est consacrée. Il peut être utilisé en conjonction avec des chaines, des fichiers ou des conteneurs. On peut aussi implanter dans une classe une fonction *dans* qui sera appelé en contexte.

Classe

Si une fonction *dans* est présente parmi les méthodes d'une classe, elle sera automatiquement appelée dès qu'un « *dans* » sera effectué sur une instance de cette classe. Si cette méthode n'est pas présente, KIFF renverra *faux*.

Opérateur

L'opérateur *dans* peut être utilisé en conjonction avec une fonction de rappel introduite par l'opérateur *avec*. Dans ce cas, la comparaison sera faite via cette fonction.

Négation : *pas* ou *non*

On peut utiliser l'opérateur *pas* (ou *non)* pour vérifier si une valeur *n'est pas dans* un conteneur.

si (i *pas dans* v) …

Exemple

Voici un premier exemple avec un dictionnaire :

```
dictionnaire dico;
table lst;
dico={'a':1,'b':6,'c':4,'d':6};

// renvoie vrai ou faux
si (6 dans dico)
    affiche("Comme attendu","\n");

//Le receveur est une table, nous renvoyons donc une table d'indexes.
lst=6 dans dico;

chaine s;
pour (s dans lst)
  affiche("LST:",s,"\n");
```

EXÉCUTE

Comme attendu
LST: b
LST: d

Comme nous le voyons sur cet exemple, le type de la variable de réception définit les données renvoyées.

Exemple avec une fonction

Dans cette fonction, i sera toujours instancié avec la valeur sur la gauche du *dans*.

```
fonction compare(nombre i, nombre j) {
        si (i<j)
                renvoie(vrai);
        renvoie(faux);
}

si (3 dans vect avec compare)
        affiche("OK");

lst=3 dans vect avec compare;

//dans notre exemple ci-dessus, i=4…
```

Exemple avec une classe

```
classe testclasse {
        nombre i;

        //le type du paramètre peut être n'importe quoi
        fonction dans(nombre j) {
                si (i==j)
                        renvoie(vrai);
                renvoie(faux);
        }
}
```

Opérateur *sur*

Cet opérateur introduit quelques notions de programmation fonctionnelle dans le langage. Cet opérateur est utilisé pour appliquer une fonction sur un conteneur. Il ressemble en cela à la méthode *applique* avec comme différence essentielle que les expressions en « sur » peuvent être emboitées.

Il existe en fait deux opérateurs « sur », l'un qui renvoie un autre conteneur et l'autre qui renvoie une valeur. La sélection de l'un ou l'autre mode se fait en fonction du contexte droit.

Contextes

Les contextes sont les contextes habituels dans KIFF, ce peut être soit la variable de réception, soit le contexte dans lequel l'expression apparait. Dans les exemples qui suivent, f0,f1 sont des fonctions et c est un conteneur.

```
nombre v= f0 sur c; //ici l'exécution doit renvoyer une valeur numérique

v= 10+(f0 sur c); //Le context de cette operation implique aussi une valeur numérique

tablenombres iv= f1 sur c; // ici le retour est un conteneur

v= f0 sur (f1 sur c); // Ce cas est le plus compliqué, f1 sur c doit renvoyer un conteneur pour que l'autre "sur" est un sens.
```

Ces exemples montrent que les deux opérateurs se différencient par leur valeur de retour, soit une valeur soit un conteneur.

La règle est très simple. Vous devez choisir en fonction du contexte la fonction qui renverra une valeur ou un conteneur. Lorsque le contexte est sous-spécifié comme dans le dernier exemple, alors la valeur de retour doit être un conteneur.

Deux sortes de fonction

Les fonctions en question ont quasiement la même signature.

Les fonctions conteneur

Ces fonctions sont appelées dans le contexte d'un retour de conteneur.

```
fonction fconteneur(itérateur it);
fonction fconteneur(itérateur it, omni s); //pour garder une trace pendant tout l'opération
```

```
fonction fconteneur(itérateur it, omni s=1);  //avec une valeur initiale
```

Le premier paramètre est toujours un itérateur, avec lequel on peut avoir accès à la clef : *it.clef()* ou à la valeur : *it.valeur()*. On peut aussi modifier la valeur courante via : *place(valeur)*.
Le second paramètre est optionnel. Il doit avoir le type « omni » de façon à pouvoir le garder « vivant » pendant toute la durée du processus. On peut initialiser dans la fonction elle-même la première valeur de cette variable. Si l'initialisation n'a pas été faite, alors le type de cette variable est celui du premier élément du conteneur.

Important: Les valeurs retournées par cette fonction sont enregistrées les unes après les autres dans le conteneur de réception. Dans le cas d'un contexte sous-spécifié, le conteneur par défaut a le même type que le conteneur courant.

Les fonctions valeur.

```
fonction fconteneur(itérateur it);
fonction fconteneur(itérateur it, omni s); //pour garder une trace pendant tout l'opération
fonction fconteneur(itérateur it, omni s=1);  //avec une valeur initiale
```

Ces fonctions sont très semblables aux précédentes, à un détail près, elles ne doivent pas renvoyer de valeur. Plus exactement, elles utilisent la variable « s » comme variable d'accumulation.

Important: *A la fin de l'application de cette fonction, c'est la variable « s » qui sera renvoyée comme résultat final. Si cette variable est absente, la fonction renvoie* vrai.

Fonctions Lambda

Une fonction lambda est en fait une fonction déclarée « sur site ». Plus exactement, il s'agit à la fois d'une déclaration de fonction et d'une application de celle-ci sur des conteneurs. Les arguments et le corps de la fonction sont donc déclarés avec le mot clef *lambda* devant l'appel à l'opérateur *sur*, seul opérateur pour lesquelles celles-ci sont évaluées. La déclaration d'une telle fonction obéit aux mêmes règles que les fonctions ci-dessus. Elle a la forme suivante :

```
lambda(itérateur it,omni b=1) {votre code…} sur conteneur;
```

renvoie(vide)

Lorsque l'une des fonctions ci-dessus renvoie « vide », l'itération est interrompue.

Exemple: Une fonction conteneur simple

Nous renvoyons un conteneur qui contient les valeurs du conteneur courant incrémentées de 1.

```
//Conteneur initial
dicocn iv={"a":1,"b":2,"c":4,"d":8,"e":16,"f":32};
dicocn résultat;

//On incrémente la valeur de 1
fonction increment(itérateur it) {
    renvoie(it.valeur()+1); //Chaque valeur sera enregistrée dans un conteneur
}

//On applique la fonction à iv
résultat=increment sur iv;
```

Le résultat est: **{'a':2,'b':3,'c':5,'d':9,'e':17,'f':33}**
Nous gardons les mêmes clefs…

Exemple: Une fonction plus compliquée

```
dicocn iv={"a":1,"b":2,"c":4,"d":8,"e":16,"f":32};
dicocn résultat;

//On multiplie chaque élément. « b » vaut 1 au début
fonction fmultiply(itérateur it,omni b=1) {
    b*=it.valeur();
    renvoie(b);
}

résultat=fmultiply sur iv;
```

Le résultat est: {'a':1,'b':2,'c':8,'d':64,'e':1024,'f':32768}

Exemple: Fonction valeur

```
//Conteneur initial
dicocn iv={"a":1,"b":2,"c":4,"d":8,"e":16,"f":32};
nombre résultat; //On range le résultat dans un nombre

//  « b » nous sert d'accumulateur avec comme valeur initiale 1.
fonction fmultiply(itérateur it,self b=1) {
    b*=it.valeur();
}

résultat=fmultiply on iv;
```

Le résultat est: 32768

Exemple: Appels emboités.

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

fonction fmultiply(itérateur it,self b=1) {
    b*=it.valeur();
```

```
}

fonction increment(itérateur it) {renvoie(it.valeur()+1); }

résultat=fmultiply sur (increment sur iv);
```

Le résultat est: 151470

Exemple: Une lambda

```
//Chaine initiale
chaine s="12:1-14:2";

//Nous appliquons notre lambda sur la decomposition de cette chaine
table tv=lambda(itérateur it) {renvoie(it.valeur().éclate(":"));} sur s. éclate("-");
```

Le résultat est: [['12','1'],['14','2']]

Langage Fonctionnel: à la Haskell

KIFF fournit un jeu particulier de d'instructions proche du langage Haskell. Ce langage est un langage fonctionnel qui fournit une façon à la fois très compacte et très puissante de représenter certains problèmes mathématiques.

Nous avons ajouté à KIFF une partie de ce pouvoir expressif de Haskell, en nous concentrant sur les fonctions de base du langage. Nous ne prétendons pas ici que KIFF agit à la manière d'un compilateur complet du langage, mais il fournit les méthodes les plus intéressantes du langage.
Nous utiliserons malgré tout le terme Haskell dans le reste de ce chapitre, en sachant malgré tout qu'il ne s'agit que d'un sous-ensemble d'un langage beaucoup plus vaste et complexe.

Avant de commencer: quelques opérateurs nouveaux.

Avant de decrire le langage de façon plus détaillé, nous allons presenter quelques nouveaux opérateurs, qui assurent une plus grande compatibilité avec Haskell. Ils sont aussi disponibles dans des programmes KIFF plus tradionnels.

Déclaration d'intervalle: [a..b]

Il est désormais possible de declarer des intervalles à la manière d'Haskell, grace à un nouvel opérateur: ".."

Par exemple [1..10] définit la table: [1,2,3,4,5,6,7,8,9,10].

1. **Le pas**

Par défaut, le pas est de 1, mais il est possible d'initialiser un pas différent, soit avec l'opérateur « : », soit en fournissant l'élément suivant dans la description.

Par exemple [1..10:2] définit la table: [1,3,5,7,9].

En fournissant l'élément suivant:

Par exemple [1,3..10] définit la table: [1,3,5,7,9].

Cela marche aussi avec des chaines de caractères:

Par exemple [‘a’,’c’..’g’] définit la table: ['a','c','e','g'].

Nous aurions aussi pu définir cette table sous la forme: [‘a’..’g’:2]...

2. **Intervalles infinis**

Haskell fournit aussi la notion d’un intervalle infini d’éléments. Il y a deux possibilities, soit en omettant le premier élément, soit en ignorant le dernier élément:

- [1..] définit an infinite table that starts at 1, forward: [1,2,3,4…
- [..1] définit an infinite table that starts at 1, backward: [1,0,-1,-2,-3…

You can also use different steps:

- [1..:2] définit une table infini qui commence à 1, ascendant: [1,3,5…
- [..1:2] définit une table inifini qui commence à 1, descendant: [1,-1,-3…

Or

- [1,3..] définit une table qui commence à 1, ascendant: [1,3,5…
- [..-1,1] définit une table qui commence à 1, descendant: [1,-1,-3…

Deux nouveaux opérateurs: *&&& et* ::

Ces deux opérateurs sont utilisés pour concaténer des listes d’éléments ensemble ou ajouter un élément à une table.

1. **Fusion: “&&&”**

Cet opérateur est utilisé pour fusionner différents éléments dans une table. Si un élément n’est pas une liste, il est simplement ajouté dans la liste courante.

```
table v= 7 &&& 8 &&& [1,2];
afficheligne(v);
```

v=**[7,8,1,2]**

Cet opérateur est semblable au “++” dans Haskell. Comme cet opérateur avait déjà une sémantique dans KIFF, nous avons préféré en introduire un nouveau.

2. **Ajoute: “::”**

Cet opérateur est semblable au precedent, à la difference près qu’il insère l’élément dans la table à gauche ou à droite de l’élément à insérer.

1::v → **[1,7,8,1,2]** *La nouvelle valeur de v*
v::12 → **[1,7,8,1,2,12]** *La nouvelle valeur de v*

Premiers pas

Déclaration d'une expression Haskell dans KIFF.

Toutes les instructions Haskell dans KIFF doivent être declarées entre "<..>", dont le compilateur KIFF se sert pour détecter une formule Haskell.

Exemple:

```
table v=<projette (+1) [1..10]>;
```

L'instruction ci-dessus ajoute « 1 » à chaque élément de la liste.

Structure la plus simple

La structure la plus simple pour un programme Haskell est de retourner une valeur:

- <1>;

Si vous

- <(3+1)>;

Dans les deux cas, une valeur atomique est renvoyée.

Exemple:

- <(12+3)> renvoie 15…

Itération

Le langage Haskell fournit une façon très pratique et efficace de représenter des listes, lesquelles sont implantées dans KIFF sous la forme de tables, de façon à assurer un échange simple entre un programme KIFF et une expression Haskell.

Une iteration dans une expression Haskell à la forme suivante:

- <x | x <- v, Booléen>

Cette expression renvoie une liste…

On peut lire cette expression sous la forme suivante:

1. *L'expression renvoie x comme valeur de liste*
2. *x est obtenu en itérant dans v*
3. *Nous plaçons une contrainte Booléenne sur x, qui peut être omise.*

Du fait de l'itération dans l'expression, le résutat renvoyé ne peut être qu'une liste.

Exemple:

<x | x <- [-5..5], x!=0> *donne* [-5,-4,-3,-2,-1,1,2,3,4,5]

Combiner

Vous pouvez combiner plusieurs iterations ensembles, soit en les encapsulant soit en les rendant simultanées.

1. Encapsulation

Les différentes iterations sont séparées par une: ","

- <x+y | x <-v, y <- vv, (x+y)>10>

2. Simultanée

Les différentes iterations sont combinées avec: ";"

- <x+y | x <-v ; y <- vv, (x+y)>10>

Exemple:

<(x+y) | x <- [1..5], y <- [1..5]> //Combinée
donne **[2,3,4,5,6,3,4,5,6,7,4,5,6,7,8,5,6,7,8,9,6,7,8,9,10]=25 éléments…**

<(x+y) | x <- [1..5] ; y <- [1..5]> //simultanée
donne **[2,4,6,8,10]=5 éléments…**

Important

Pour renvoyer le résultat d'une opération entre éléments, n'oubliez pas de les mettre entre parenthèses: (x+y)

Patron de table

Vous pouvez utiliser des patrons de table pour extraire des éléments d'une liste, quand cette liste est composée de sous-listes.

Exemple

```
table v=[[1,"P",vrai],[2,"C",faux],[3,"E",vrai]];
table vv=<[y,t] | [y,n,t] <- v, y>1>;
```

donne: **[[2,faux],[3,vrai]]**

Itération bis

KIFF fournit aussi un type itérateur que l'on peut utiliser dans des expressions Haskell:

- **<x.valeur() | itérateur x=v, x.valeur()!=0>**

Exemple:

```
<x.clef() | itérateur x=[-10..20], x.valeur()%2==1>
```

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

Déclaration d'une variable locale

Il existe différente façon de déclarée une variable dans une expression Haskell.

Important : Toutes sont déclarées comme étant de type *omni*. Ce qui signifie qu'elles ne prendront leur valeur et leur type qu'à l'exécution.

1. **Opérateur : soit**

Vous pouvez utiliser l'opérateur « soit » pour associer une variable ou un patron de tables avec une expression :

- <a | soit a=10>;
- <(a+b+c) | soit [a,b,c] = [1,2,3]>;

« soit » s'utilise de deux façons différentes : Si la valeur de retour est déjà déclarée dans l'expression, alors on peut enchaîner plusieurs « soit », séparée par une virgule.

- <(a+b) | soit a=10,soit b=20>

Sinon, on peut encore renvoyer la valeur via un « dans » et dans ce cas, on ne peut utiliser qu'un seul « soit » à la fois, les instanciations sont alors séparées par une virgule.

- <soit a=10,b=20, dans (a+b)>

A noter la virgule avant le « dans » qui n'est pas dans la version officielle de Haskell, mais qui est nécessaire dans KIFF pour correctement compiler l'expression.

Quand une itération est déclarée dans l'expression, le « soit » est réévalué à chaque passage.

- <a | x <- [1..10], soit a=x*2>

2. Opérateur « où »

L'opérateur « où » est utilisé pour déclarer des variables globales. Il est placé à la fin de l'expression Haskell. Son évaluation n'est faite qu'une seule fois avant toutes analyses.

- <a | soit a=w+10, où w=20;>

Il peut y avoir avant de déclarations que nécessaire. Notez que chaque déclaration doit se terminer par un « ; ».

- <a | soit a=w1*w2, où w1=20;w2=30;>

Fonctions

Le langage Haskell permet aussi la déclaration de fonctions, lesquelles peuvent être appelées depuis KIFF. Notez que chaque argument est déclaré sous la forme d'un type « omni ».

Déclaration

Une fonction se déclare de la façon suivante:

- <nom(a1,a2…) : expression Haskell>;

Elles peuvent être appelées depuis Haskell comme suit: nom(p1,p2…).

Exemples:

```
<un(x) : (x+1)>;
nombre val=un(12);
```

val est: **15**

```
<plusun(v) : (x+1) | x <- v>;
```

```
table vect=plusun([1..10]);
```

vect est: **[2,3,4,5,6,7,8,9,10,11]**

Implantation

Chaque fois que vous déclarez une fonction Haskell, celle-ci est déclarée sous la forme d'une fonction KIFF dont chaque élément est une variable de type « omni ».

Ainsi la déclaration "plusun" correspond à:

```
fonction plusun(omni v) {…}
```

De plus, les arguments de ces fonctions peuvent être aussi bien des valeurs atomiques (nombre, réel ou chaine) que des déclarations de tables.

Garde

Le langage Haskell fournit un mécanisme qui ressemble beaucoup à l'opérateur « parmi » : la *garde.* Une garde est une succession de tests associés avec une action. Chaque test est introduit avec un « | ». La valeur par défaut est elle introduite avec le mot clef « autrement ».

Exemple:

```
<imb(bmi) : | bmi<=10 = "petit" | bmi<=20 = "moyen" | autrement = "grand">;
```

imb(12) donne "moyen" comme réponse.

Déclarations multiples

Il est possible d'éclater la déclaration d'une fonction en une succession d'expressions Haskell. Dans ce cas, la liste des arguments peut contenir des valeurs atomiques. Lorsque l'expression est évaluée, les paramètres des appels sont comparés aux arguments de chacune des déclarations. Si la comparaison échoue, le système passe à l'expression suivante.

Exemple:

```
<fibonacci(0) : 0>;
<fibonacci(1) : 1>;
<fibonacci(n) : a | soit a=fibonacci(n-1)+fibonacci(n-2)>;
```

fibonacci(10) est 55

Lorsque n vaut 1 ou 0, la fonction correspondante est exécutée pour renvoyer le résultat adéquat.

arrête

La fonction « arrête » peut être utilisée pour faire échouer une déclaration courante, de façon à forcer l'exécution de la déclaration suivante.

Exemple:

```
<maboucle(v): si (v.taille()>10) arrête sinon v[0]>;
<maboucle(v): v[10]>;

<maboucle([1..10])> donne 1, la taille de la liste est 10

<maboucle([1..20])> donne 11, la taille de la liste est 20
```

Itérer sur les listes

Haskell peut itérer sur les listes de la même façon que Prolog. On peut pour se faire utiliser soit l'opérateur « | » comme en Prolog ou l'opérateur « : » comme en Haskell pur. Il faut noter d'ailleurs la similitude sémantique dans ce cas avec le concaténateur de liste « : ».

Exemple:

```
<inverser([ ]) : "vide">;
<inverser([premier:reste]) : [a,premier] | soit a = inverser(reste)>;

inverser(['a'..'e']);
```

donne **[[[[[vide','e'],'d'],'c'],'b'],'a']**

Notez la présence d'un « ; » à la fin de chaque ligne.

Appeler une fonction

Vous pouvez appeler n'importe quelle fonction ou méthode depuis une expression Haskell. On peut alors panacher les appels à la mode Haskell ou à la mode KIFF dans la même expression. Voici par exemple, l'application de la méthode « rogne » sur chaque élément d'une liste de chaines.

- `<rogne1(w) : x | soit x=w.rogne()>; //Le plus simple`
- `<rogne2(w) : x | soit x=<rogne w>>; //pur appel Haskell`

```
//Nous définissons une fonction
fonction Rogne(chaine c) {
    renvoie(c.rogne());
}
```

- **<rogne3(w) : x | soit x=Rogne(w)>; //Appel direct à la fonction**
- **<rogne4(w) : x | soit x=<Trim w>>; //à la Haskell**

N'importe quelle fonction ou méthode peut être appelée à condition qu'elle corresponde au type de la variable.

- **<ajouter(v) : <somme v>>;**

Il n'existe aucune différence entre un appel direct et un appel à la Haskell.

Exemple: Trier une liste

```
<trirapide([ ]) : [ ]>;  //liste vide on arrête là
<trirapide([fv|v]) :  (mn &&& fv &&& mx) | //nous fusionnons nos sous-listes
    soit mn = trirapide(<a | a <- v, a<=fv>), //les éléments les plus petits
    soit mx = trirapide(<a | a <- v, a >fv>)>;//les éléments les plus grands
```

Opérations

Haskell fournit un ensemble de fonctions spécifiques qui ne peuvent être utilisées qu'au sein d'expressions Haskell. Certaines appliquent des filtres ou des opérations sur des listes. D'autres effectuent des cycles ou des duplications.

<prend nb liste>

Cette fonction retourne les nb premiers éléments d'une liste. Ainsi, si vous effectuez une boucle sur une liste infinie, elle peut être utilisée pour bloquer l'itération quand un certain nombre d'éléments ont été extraits.

Exemple

<prend 10 [1,5..100]> donne [1,5,9,13,17,21,25,29,33,37]

<abandonne nb liste>

Cette fonction vous rend tous les éléments à l'exception des nb premiers.

Exemple

<abandonne 10 [1,5..100]> donne [41,45,49,53,57,61,65,69,73,77,81,85,89,93,97]

<cycle liste>

Cette méthode permet d'effectuer des cycles dans une liste.

Exemple

v=<prend 10 <cycle [1,2,3]>> donne [1,2,3,1,2,3,1,2,3,1]

<répète valeur>

Cette fonction crée une liste dans laquelle valeur est répétée ad infinitum.

Vous pouvez combiner cette instruction avec "prend" pour limiter le nombre d'itérations.

Exemple

v=<prend 10 <répète 5>> donne [5,5,5,5,5,5,5,5,5,5]

<réplique nb valeur>

Cette fonction réplique une valeur en une liste de nb éléments.

Exemple

<réplique 3 [10]> donne [[10],[10],[10]]

Composition: "."

Vous avez certainement noté que « prend » prend le contrôle du déroulement des opérations à sa gauche. On appelle ce mécanisme la *composition.* Il permet à un programme de placer une limite sur ce que les niveaux inférieurs font. De façon à simplifier l'écriture, on peut utiliser l'opérateur « . ».

Ainsi, la formule **<prend 10 <répète 5>>** peut aussi s'écrire:

<prend 10 .répète 5>

<projette (op) liste>

Cette fonction est utilisée pour appliquer une fonction ou une opération à chacun des éléments d'une liste. Si « op » est réduit à un simple opérateur, alors chaque élément est combiné avec lui-même. Vous pouvez aussi utiliser une lambda de la forme : (\x -> …)

1. **Avec un opérateur simple**

   Si un opérateur simple est fourni, on combine chaque élément avec lui-même via cet opérateur.

- **<projette (+) [1..10]> donne [2,4,6,8,10,12,14,16,18,20]**
(1+1/2+2/3+3/4+4/5+5/6+6/7+7/8+8/9+9/10+10)

2. Avec un opérateur et une valeur

On fournit ici à la fois l'opérateur et une valeur. Attention, la position de la valeur est importante dans l'expression :

- **<projette (-1) [1..10]> donne [0,1,2,3,4,5,6,7,8,9]**
(1-1/2-1/3-1/…/10-1)

En revance:

- **<projette (1-) [1..10]> donne [0,-1,-2,-3,-4,-5,-6,-7,-8,-9]**
(1-1/1-2/1-3/1-4/…1-10)

3. Avec une lambda

Une lambda en Haskell est définie de la façon suivante :: (\x_0 x_1..x_n -> x_0+x_1+..+x_n), où x_0 x_1..x_n sont les arguments de la lambda, suivi de "->" et une expression particulière :

Dans le cas d'un map, la lambda n'a qu'un seul argument :

- **<projette (\x -> (x+4)/3) [1..10]>** donne **[1,2,2,2,3,3,3,4,4,4]**

4. Avec une fonction

Vous pouvez aussi appliquer une fonction à chaque élément :

- **<projette (cos) [0,0.1..0.4]> donne:**
[1,0.995004,0.980067,0.955336,0.921061]

Cette fonction peut être une fonction Haskell, ou une méthode ou une fonction KIFF.

```
fonction Min(réel y) {
    renvoie(y-1);
}
```

<projette (Min) [1..10]> donne [0,1,2,3,4,5,6,7,8,9]

<filtre (condition) liste>

filtre est utilisé pour filtrer les éléments d'une liste ayant une propriété particulière. Cette propriété peut être exprimée avec un opérateur de comparison, avec une lambda ou une fonction qui renvoie *vrai* ou *faux*.

Exemples

1. On ne garde que les valeurs > 3:

➢ **<filtre (>3) [1..10]> donne [4,5,6,7,8,9,10]**

2. Dans l'exemple ci-dessous, on ne garde que les valeurs paires, via une lambda.

➢ **<filter (\x -> (x%2)==0) [1..10]> donne [2,4,6,8,10]**

3. On peut aussi utiliser une fonction pour la comparaison:

```
fonction impair(nombre x) {
        si (x%2==0)
                renvoie(faux);
    renvoie(vrai);
}
```

➢ **<filtre (impair) [1..10]> donne [1,3,5,7,9]**

4. Enfin, on peut composer notre expression avec un « projette ».

➢ **<filtre (impair) . projette (*3) [1..10]> donne [3,9,15,21,27]**

<prendJusquà (condition) liste>

prendJusquà place une condition sur chaque élément qui sort de la liste. Il cesse l'itération dès que la valeur ne correspond plus à la condition. Il fonctionne de la même façon que « prend », mais au lieu de compter les éléments, il place une condition.

Exemples

1. Itération sur une liste infinie

➢ **<prendJusquà (<100) [1,11..]> donne [1,11,21,31,41,51,61,71,81,91]**

Dès que la valeur a dépassée 100, l'itération a été bloquée.

2. Combiner un projette et un filtre

Dans cet exemple, nous extrayons toutes les valeurs impaires dont le carré est <500.

➢ **<filtre (impair) . prendJusquà (<500) . projette (*) [1..]>**

Le résultat: **[1,9,25,49,81,121,169,225,289,361,441]**

<abandonneJusquà (condition) liste>

Cette fonction abandonne tous les éléments qui correspondent à la condition, puis ensuite renvoie tous les autres, dès que cette condition échoue.

Exemple

<abandonneJusquà (estchiffre) "12345ABCD123" > donne ABCD123

<fusionne l1 l2..ln>

Fusionne différente listes ensembles, chaque élément renvoyé est une liste contenant un élément de l1, l2..ln.

Exemples

- **<fusionne [0..2] [0..2] [0..2]>**

  Le résultat est: **[[0,0,0],[1,1,1],[2,2,2]]**

<fusionneAvec (f) l1 l2 l3…ln>

fusionneAvec combine differentes listes ensemble via *f*. Si *f* est une lambda, alors son nombre d'arguments doit correspondre au nombre de liste.

Exemples

1. Combiner trois listes avec "+"

- **<fusionneAvec (+) [0..10] [0..10] [0..10]>**

  Résultat: **[0,3,6,9,12,15,18,21,24,27,30]**

2. Avec une lambda

- **<fusionneAvec (\x y z -> x*y+z) [0..10] [0..10] [0..10]>**

  Résultat: **[0,2,6,12,20,30,42,56,72,90,110]**

3. Composition avec des listes infinies

- **<prendJusquà (<100) .fusionneAvec (\x y z -> x*y+z) [0..] [0..] [0..]>**

  Résultat: **[0,2,6,12,20,30,42,56,72,90]**

<plieg|plied (f) premier liste>

Ces opérateurs appliquent une fonction, une lambda ou une operation sur une liste, avec “premier” comme valeur initiale. La fonction lambda doit avoir deux arguments. La différence entre “plieg” and “plied” est la direction du “repliage”. “plieg” commence par le début de la liste, tandis que « plied » commence par la fin. « plieg » traverse la liste en avant, et « plied » par l’arrière.

Dans une expression lambda, le dernier argument est l’élément de la liste dans le cas d’un « plieg », le premier pour « plied ». L’autre élément est l’accumulateur dont la valeur finale sera renvoyée par l’expression.

Ces fonctions renvoient une valeur et non une liste.

Exemples

1. Sommer les éléments d’une liste avec 100 comme valeur initiale :

- **<plieg (+) 100 [1..10]> donne 155… (100+1+2+3…+10)**

2. Lambda pour un pliage par la gauche

- **<plieg (\ acc x -> acc+2*x) 10 [1..10]> donne 120**

3. Lambda pour un pliage par la droite

- **<plied (\ x acc -> acc+2*x) 10 [1..10]> donne 120**

  Notez que l’élément de la liste: “x” est le premier élément de la lambda.

<plieg1|plied1 (f) liste>

Ces deux fonctions sont identiques aux deux autres, mais elles prennent comme valeur initiale le premier élément de la liste.

Exemples

1. Somme des éléments d’une liste, la première valeur est 1…

- **<plieg1 (+) [1..10]> donne 55… (1+2+3…+10)**

2. Avec une lambda

- **<plieg1 (\ acc x -> acc+2*x) [1..10]> donne 109**

Notez que l'élément de la liste: "x" est le second élément de la lambda.

3. Avec une lambda pour plied1

- **<plied1 (\ x acc -> acc+2*x) [1..10]> donne 110**

  Notez que l'élément de la liste: "x" est le premier élément de la lambda.

scang,scand,scang1,scand1

Ces fonctions sont identiques aux précédentes, à une différence près : elles stockent les résutltats intermédiaires dans des listes.

Exemples

1. Somme des éléments d'une liste, la première valeur est 1…

- **<scang1 (+) [1..10]> donne [3,6,10,15,21,28,36,45,55]**

2. Avec une lambda

- **<scang1 (\ acc x -> acc+2*x) [1..10]> donne [5,11,19,29,41,55,71,89,109]**

  Notez que l'élément de la liste: "x" est le second élément de la lambda.

1. Avec une lambda mais avec scand

- **<scand1 (\ x acc -> acc+2*x) [1..10]> donne [100,98,94,88,80,70,58,44,28]**

  Notez que l'élément de la liste: "x" est le premier élément de la lambda.

Variables avec fonctions: Fonctions de rappel

L'opérateur « avec » a déjà été décrit dans les sections précédentes. Il est possible d'associer une fonction à une variable ou à une classe. Cette fonction de rappel est automatiquement appelée dès que la valeur de cette variable change. Selon les objets, cette fonction de rappel peut prendre différents paramètres. Pour les cas de base, elle en prend deux : la valeur avant modification, la valeur après modification. Pour les classes, elle en prend trois, le premier étant la classe elle-même.

Important

Après l'opérateur *avec,* il est possible à la place du nom de la fonction, d'utiliser une variable de type *appel.* Dans ce cas, cette variable peut être modifiée avec d'autres noms de fonctions.
Lorsque l'appel sera effectué, ce sont ces derniers noms qui seront utilisés. De cette façon, on peut utiliser des fonctions de rappel dynamiques.

Initialisation de valeurs

Si l'on veut initialiser une variable déclarée avec une fonction de rappel, il faut placer cette initialisation après la déclaration de la fonction, comme suit :

nombre i avec fonc=100;

Exemple avec une variable

```
//Nous implémentons une fonction de rappel
fonction react(omni before,omni after) {
        afficheligne("dico",before,after);
}
//associé avec un dictionnaire
dictionnaire dico avec react;

//quand dico est modifié, "react" est appelé
dico={'a':1,'b':6,'c':4,'d':6};
```

Exemple avec une classe

```
//Nous implémentons notre classe anoeud
classe anoeud {
        nombre i;
```

```
        fonction chaine() {
                renvoie(i);
        }
        fonction seti(nombre j) {
                afficheligne("J=",i,j);
                i+=j;

        }
}

//Une classe connexe
classe connexe {
        table v;
}

//la fonction de rappel, le premier paramètre est un élément de type connexe
fonction transconnexe(connexe f,nombre before,nombre after) {
        itérateur it=f.v;
        //nous iterate on chaque élément de notre table
        pour (it.commence() tantque it.nfin() faire it.suivant())
                it.valeur().seti(before+after);
}

//Le reste de notre classe
classe connexe {
        //i est associé avec notre nouvelle fonction de rappel
        nombre i avec transconnexe;

        fonction _initiale(nombre j) {
                i=j;
        }

        fonction addanoeud(nombre j) {
                anoeud xn;
                xn.i=j+10;
                v.empile(xn);
        }
}

//La fonction transconnexe est automatiquement appelé, chaque fois que i est
modifié.
connexe c(10); //sera appelé ici
c1.i=100; //et ici
```

Synchronisation

La synchronisation des tâches est un problème complexe pour lequel KIFF fournit un certain nombre de solutions.

1. chaine s=attends(chaine,chaine,chaine): cette instruction place la tâche en attente sur une série de chaines. Lorsque une de ces chaines sera activée via *lance*, alors la tâche reprendra son exécution.

2. libère(chaine): Cette instruction réactive les tâches en attente sur *chaine*.

3. libère(): Cette instruction réactive toutes les tâches en attente.

4. tue(chaine): cette instruction tue toutes les tâches en attente sur chaine.

5. tue(): cette instruction tue toutes les tâches en attente.

6. enattente():cette instruction renvoie une table de toutes les chaines d'en attente.

7. verrouille(chaine s): Cette instruction place un verrou qui bloque l'accès des autres tâches à ces ressources.

8. déverrouille(chaine s): Cette instruction retire un verrou qui bloque l'accès des autres tâches à ces ressources.

9. synchrone: Cette fonction doit être associée à une déclaration de variable via « avec ». IMPORTANT : la variable doit avoir une interprétation Booléenne : *vrai ou faux.*

10. attendsquefaux(var): Cette fonction met une tâche en attente jusqu'à ce que la valeur de la variable ait la valeur *faux*. Cette fonction ne peut prendre comme valeur que des variables ayant été déclarées comme *synchrone*.

11. attendsfinjoindre(): Cette fonction permet d'attendre la fin de l'exécution de toutes les tâches ayant été déclarées avec le type *joindre*.

- Exemple:

```
tâche  compte(nombre i) {
        nombre j=10;
        i+=1;
        i+=1;
        i+=1;
        j+=i;
        afficheligne("j=",j);
        attends("Here");
        afficheligne("I=",i,j);
}

tâche comptebis(nombre i) {
        nombre j=10;
        i+=1;
        i+=1;
        i+=1;
        j+=i;
        afficheligne("bis j=",j);
        attends("Here");
        afficheligne("bis I=",i,j);
}

compte(5);
comptebis(10);
afficheligne("Nous revenons");
libère("Here");
```

Exécution

Après exécution nous obtenons les lignes suivantes:

j= 18
bis j= 23
Nous revenons
I= 8 18
bis I= 13 23

Mutex: verrouille et déverrouille

Ces fonctions permettent de bloquer l'accès concurrent à certaines lignes de code par des tâches différentes.

Exemple

```
//Nous implémentons notre tâche
tâche lancement(chaine n,nombre m) {
        nombre i;
        afficheligne(n);
        //nous affichons nos valeurs
        pour (i=0;i<m;i++)
```

```
                affiche(i," ");
        afficheligne();
}

fonction principal() {
        //nous lançons nos tâches
        lancement("Premier",2);
        lancement("Second",4);
}
```

Après exécution nous obtenons un affichage aléatoire, car l'ordre d'exécution des tâches est décidé au niveau du noyau système et non de notre programme.

```
PremierSecond
00  1 1
2 3
```

En revanche, grâce à des verrous, nous pouvons imposer un certain ordre.

```
//Nous re-implémentons notre tâche avec un verrou
tâche lancement(chaine n, nombre m) {
        verrouille("lancement"); //Nous verrouillons ici
        nombre i;
        afficheligne(n);
        //nous affichons all notre valeurs
        pour (i=0;i<m;i++)
                affiche(i," ");
        afficheligne();
        déverrouille("lancement"); //Nous déverrouillons avec la même chaine
}
```

En exécutant le code ci-dessus, nous obtenons un affichage totalement différent.

```
Premier
0 1
Second
0 1 2 3
```

Important:

Les verrous sont globaux à l'ensemble du programme. Ce qui signifie qu'un verrou peut être placé par une fonction mais déverrouillé par une autre.

- Tâches *protégées*

L'exemple ci-dessus aurait pu être réécrit avec l'attribut de tâche : *protégée*.

```
//Nous re-implémentons notre tâche comme tâche protégée
protégée tâche lancement(chaine n, nombre m) {
        nombre i;
        afficheligne(n);
        //nous affichons nos valeurs
        pour (i=0;i<m;i++)
                affiche(i," ");
        afficheligne();
}
```

Cette fonction aura le même comportement que précédemment. En fait, une tâche protégée introduit un verrou à son lancement et le retire en fin d'exécution.

Sémaphores: attendsquefaux et synchrone.

La synchronisation de fonction doit souvent se faire sur d'autres critères que ceux présentés jusque là. Dans un contexte multi-tâche, Il peut être utile de bloquer ou déclencher le fonctionnement d'une tâche en fonction de la valeur d'une variable. Cette variable peut avoir tous les types possibles à condition d'avoir une interprétation booléenne. Par exemple, si c'est une chaine, la chaine vide sera considérée comme étant la valeur Booléenne *faux*. Si c'est un nombre, on renverra *faux*, s'il vaut 0.

- …avec synchrone

Il faut nécessairement déclarer notre variable avec une fonction système particulière: *synchrone*.

- attendsquefaux(var);

Cette méthode placera notre tâche en attente, tant que la *variable synchrone* n'aura pas la valeur *faux*.

Exemple

```
//D'abord nous déclarons une variable stopby comme synchrone
//Important: sa valeur initiale doit être différente de 0
nombre stopby avec synchrone=1;

//Nous implémentons notre tâche
tâche lancement(nombre m) {
        //nous initialisons stopby avec le nombre de boucles
        stopby=m;
```

```
        nombre i;
        //nous affichons toutes nos valeurs
        pour (i=0;i<m;i++) {
                affiche(i," ");
                //nous décrementons stopby
                stopby--;
        }
}

fonction principal() {
        // lancement de notre tâche
        lancement(10);
        //nous attendons que stopby vaille 0...
        attendsquefaux(stopby);
        afficheligne("Fin ");
}

principal();
```

EXÉCUTE

L'affichage du mot fin n'aura lieu qu'une fois l'ensemble des tâches exécutées.

0 1 2 3 4 5 6 7 8 9 Fin

Si nous retirons attendsquefaux, la sortie sera différente:
Fin 0 1 2 3 4 5 6 7 8 9

Ici fin sera affiché avant même que la première tâche ne soit lancée.

attendsquefaux synchronises principal et lancement ensemble.

Note

L'exemple ci-dessus aurait pu être implémenté avec *attends* et *libère*.

```
//Nous implémentons notre tâche
tâche lancement(nombre m) {
        nombre i;
        //nous affichons all notre valeurs
        pour (i=0;i<m;i++)
                affiche(i," ");
        libère("fin");
}

fonction principal() {
        //nous lancement notre tâche
        lancement(10);
        attends("fin");
        afficheligne("Fin");
}
```

```
principal();
```

attendsfinjoindre() et attribut de tâche joindre

Si nous voulons que notre programme ne reprenne la main qu'une fois toutes ses tâches exécutées, Il faut utiliser cette méthode et cette option.

Exemple

```
//Déclaration de notre tâche comme joindre
joindre tâche jaffichage(chaine s) {
            affiche(s+"\r");
}

//laquelle est lancée depuis une autre tâche, elle-même de ce type
joindre tâche lancement(nombre x) {
    nombre i;
    pour (i=0;i<5000;i++) {
         chaine s="Tâche:"+x+"="+i;
        jaffichage(s);
  }
  //nous attendons la fin de nos tâches
    attendsfinjoindre();
    affichelIgne("Fin:"+x);
}

// lancement de deux méthodes
lancement(0);
lancement(1);
//nous attendons que toutes finissent…
attendsfinjoindre ();
affichelIgne("Termination");
```

Moteur d'inférence

KIFF intègre un moteur d'inférence que l'on peut facilement mélanger avec des instructions KIFF traditionnelles.

Ce moteur est très proche dans son fonctionnement de Prolog, avec cependant quelques différences:

a) Si'l'on veut appeler un prédicat au sein d'une fonction ou d'une classe KIFF, il faut le declarer avec le type « prédicat » de façon à le distinguer des fonctions.

b) Les variables d'inférence n'ont pas besoin d'être déclarées. Cependant, elles ont une forme particulière qui les distinguent des autres variables KIFF, mais aussi des variables traditionnelles Prolog. Leur nom doit être nécessairement précédé d'un « ? ».

c) Vous pouvez ou non déclarer les termes avant leur utilisation au sein d'un prédicat. Si vous utilisez un terme sans le déclarer, alors son nom doit être précédé d'un « ? » comme pour les variables d'inférence.

N.B. Pour une description plus précise du langage Prolog, veuillez consulter des documentations appropriées.

Types

KIFF propose trois types spécifiques pour gérer les prédicats:

prédicat

Ce type permet la déclaration de prédicats si leur utilisation au sein d'instructions KIFF peut porter à confusion.

Ce type offre les méthodes suivantes :

1. nom(): *renvoie le nom du prédicat*
2. taille(): *renvoie son arité ou nombre d'arguments.*
3. _trace(bool): *active ou désactive la trace pour ce prédicat.*

terme

Ce type est utilisé pour déclarer des termes utilisés au sein des clauses.

varprédicat

Ce type est utilisé pour gérer les prédicats de façon à pouvoir explorer leurs arguments et leurs valeurs. Un *varprédicat* peut être vu comme une table, dont les arguments sont accessibles via leur position dans la liste des arguments.

Ce type offre les méthodes suivantes :

1. dictionnaire(): *renvoie le prédicat sous la forme d'un dictionnaire: ['nom':nom,'0':arg0,'1':arg1…]*
2. enregistre(): *enregistre le prédicat en mémoire*
3. enregistre(db): *enregistre les valeurs du prédicat dans la base de données.*
4. interroge(prédicat|nom,var1,var2…) : construit le prédicat correspondant et interroge la base de connaissance.
5. nom(): *renvoie le nom du prédicat*
6. retire(): *retire le prédicat de la mémoire*
7. retire(db): *détruit le prédicat au sein de la base de données*
8. table(): *renvoie le prédicat sous la forme d'une table: [nom,arg0,arg1…]*
9. taille(): *renvoie le nombre d'arguments*

Il faut noter que la méthode *prédicat*, lorsqu'elle est appelée depuis une table ou un dictionnaire, transforme le contenu de ceux-ci en un prédicat à condition que leur contenu soit semblable aux formes ci-dessus.

Exemple:

```
table v=['femme','mary'];
varprédicat fem;

fem=v.prédicat(); //nous transformons notre table en prédicat.
fem.enregistre(); //nous le stockons dans la base de faits.

v=fem.interroge(femme,?X); //Construction d'une requête de prédicat
```

```
v=fem.interroge(femme,'mary'); //avec une valeur imposée
```

Autres types d'inférence: *liste et dictionnaire associatif*

- KIFF fournit aussi les listes traditionnelles à la Prolog, avec leur opérateur particulier de décomposition de liste : « | ». Un exemple est donné plus loin pour monter comment cet opérateur peut s'utiliser.
    - Exemple

```
prédicat alist;

//iDans cette clause, C est le reste de la liste...
alist([?A,?B|?C],[?A,?B],?C) :- vrai;

v=alist([1,2,3,4,5],?X,?Y);

afficheligne(v); ➔ [alist([1,2,3,4,5],[1,2],[3,4,5])]
```

- KIFF fournit aussi un dictionnaire associatif, implémenté sous la forme d'un dictionnaire KIFF, mais dont l'ordre des arguments est important.
    - Exemple:

```
prédicat assign,avalue;


avalue(1,1) :- vrai;
avalue (10,2) :- vrai;
avalue (100,3) :- vrai;
avalue ("fin",4) :- vrai;


assign({?X:?Y,?Z:?V}) :- avalue (?X,1), avalue (?Y,2), avalue (?Z,3), avalue (?V,4);
table v=assign(?X);

afficheligne(v);        ➔ [assign({'100':'fin','1':10})]
```

Ainsi que vous pouvez le voir sur cet exemple, les *clefs* et les *valeurs* peuvent chacune dépendre d'une variable d'inférence. Cependant, il faut noter aussi que l'ordre dans lequel les *clef :valeur* sont déclarées est important. Ainsi **{?X:?Y,?Z:?V}** est différent de **{?Z:?V,?X:?Y }.**

Clauses

Une clause est définie sous la forme suivante:

prédicat(arg1,arg2…,argn) :- prédicat(arg…),prédicat(arg,…), etc. ;

Base de faits

Un fait est déclaré de la façon suivante:

```
prédicat(val,val) :- vrai;
```

Si vous remplacez “*vrai*” avec “*faux*”, l’instruction aura comme effet de retirer le fait de la base.

N.B. On peut aussi déclarer un fait de la façon suivante:

```
prédicat(val,val).
```

pour une meilleure compatibilité avec les programmes Prolog existant. Veuillez noter l’utilisation de « . » à la place de « ; » dans ce cas.

Coupe-choix et échec

KIFF fournit à la fois le traditionnel coupe-choix, sous la forme de « ! » et *échec* qui force l’échec de l’exécution d’un prédicat.

Persistence

La persistence est effectuée grâce à une base SQLite, dont la déclaration est la suivante dans un programme.

Exemple:

```
sqlite db;
db.ouvre(‘mydb’);
```

Déclaration

Pour rendre un fait persistent, il faut en avertir la base de données grâce à la méthode suivante fournie par *sqlite*: *prédicat*.

```
db.prédicat(nom,arité);
```

L’arité est le nombre d’arguments selon la terminologie traditionnelle de Prolog, d’un prédicat donné. Cette méthode crée une table SQL, dont le nom est « nom_arité », constitué d’autant de colonnes que l’arité, portant les noms P0, P1 etc…

Opérateurs

Pour enregistrer et rendre persistent un fait, on dispose des méthodes suivantes:

a) enregistre(db), qui enregistre le fait dans la base de données**.**

b) lit(db), qui récupère dans la base db un prédicat donné.

c) retire(db), qui retire un prédicat de la base db.

Exemple:

```
femme('mary') :- enregistre(db); //nous enregistrons ce prédicat dans notre base.
```

Lancer un évaluation

On lance une évaluation exactement comme un appel de fonction. Vous pouvez fournir autant de variables d'inférence que vous le désirez, mais vous ne pouvez lancer qu'une seule évaluation à la fois, ce qui impose qu'une expression complexe doit d'abord être définie sous la forme d'une clause avant d'être évaluée.

Important

Si la variable de réception e st une table, alors toutes les analyses possibles seront effectuées. Dans tout autre cas, l'évaluation s'arrêtera à la première solution trouvée.

Fonction de rappel

Un prédicat peut être déclaré avec une fonction d'appel dont la signature est la suivante:

```
fonction SiSuccès(prédicatvar p, chaine s) {
    afficheligne(s,p);
    renvoie(vrai);
}

chaine s="Parent:";

prédicat parent(s) avec SiSuccès;

parent("John","Mary") :- vrai;
parent("John","Peter") :- vrai;

parent(?X,?Y);
```

Cette fonction doit être associée avec le prédicat qui sera évalué. Cette fonction sera appelée chaque fois que l'évaluation de *parent* est réussie. Le deuxième argument correspond à celui déclaré avec *parent*.

Si cette fonction renvoie *vrai*, alors le moteur d'inférence essaie d'autres solutions, sinon il s'arrête.

Résultat:

Si nous évaluons l'exemple suivant, nous obtenons:

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

Erreurs les plus courantes avec les variables KIFF.

Les variables communes utilisées dans les prédicats, telles que *chaine, nombre ou décimal,* sont utilisées à des fins de comparaison et ne peuvent donc pas être modifées directement. Un exemple permettra de clarifier cette situation :

Exemple 1

```
chaine s="test";
chaine sx="autre";
prédicat comp;
comp._trace(vrai);

comp(s,3) :- afficheligne(s);
comp(sx,?X);
```

Exécution:

r:0=comp(s,3) --> comp(autre,?X172) --> échec

Cette clause a échouée, parce que *s* et *sx* ont des valeurs différentes.

Exemple 2

```
chaine s="test";  //Ils ont désormais la même valeur
chaine sx="test";
prédicat comp;
comp._trace(vrai);

comp(s,3) :- afficheligne(s);
comp(sx,?X);
```

Exécution:

r:0=comp(s,3) --> comp(test,?X173)
e:0=comp(test,3) --> afficheligne(s)test

succès:1=comp('test',3)

Soyez attentif lorsque vous écrivez vos clauses de bien utiliser les variables externes à des fins de comparaison et non d'instanciation. Si

vous voulez passer une chaine à votre prédicat, la variable de réception devra être une variable de prédicat.

Exemple 3

```
chaine sx="test";
prédicat comp;
comp._trace(vrai);

comp(?s,3) :- afficheligne(?s);
comp(sx,?X);
```

Exécution:

```
r:0=comp(?s,3) --> comp(test,?X176)
e:0=comp('test',3) --> afficheligne(?s177:test)test

 succès:1=comp('test',3)
```

Fonctions

Kif fournit aussi les fonctions de base de Prolog :

Fonction assertea(pred(…))

Cette fonction rajoute un fait dans la base de connaissance au début de celle-ci. Attention, on ne peut utiliser une telle fonction qu'au sein d'une déclaration de clause.

assertez(pred(…))

Cette fonction rajoute un fait dans la base de connaissance à la fin de celle-ci. Attention, on ne peut utiliser une telle fonction qu'au sein d'une déclaration de clause.

rétracte(pred(…))

Cette fonction retire un fait de la base de connaissance. Attention, on ne peut utiliser une telle fonction qu'au sein d'une déclaration de clause.

rétractetout(pred)

Cette fonction retire tous les faits relatifs à un prédicat de la base de connaissance. Si aucun nom n'est fourni, alors la base entière est vidée. Attention, on ne peut utiliser une telle fonction qu'au sein d'une déclaration de clause.

affichetouslesprédicats(pred)

Sans paramètre, cette fonction renvoie l'ensemble des prédicats présent en mémoire sous la forme d'une table. Si vous fournissez un nom, alors seuls les prédicats correspondants seront renvoyés.

Exemple

```
//Notez qu'il faut declarer "parent" au préalable pour utiliser assertea
prédicat parent;

ajout(?X,?Y) :- assertea(parent(?X,?Y));

ajout("Pierre","Roland");

afficheligne(affichetouslesprédicats(parent));
```

Quelques exemples

Tour de Hanoï

La résolution des tours de Hanoï sous forme de prédicat.

```
//Déclaration des prédicats
prédicat déplace;

//Notez les noms de varaibles qui commencent avec "?"
déplace(1,?X,?Y,_) :-
   afficheligne('Déplace le disque supérieur de ',?X,' vers ',?Y);

déplace(?N,?X,?Y,?Z) :-
   ?N>1,
   ?M is ?N-1,
   déplace(?M,?X,?Z,?Y),
   déplace(1,?X,?Y,_),
   déplace(?M,?Z,?Y,?X);

//Résultat dans res
prédicatvar res;

res=déplace(3,"gauche","droit","centre");

afficheligne(res);
```

Après exécution:

```
Déplace the le disque supérieur de  gauche  to  droit
Déplace the le disque supérieur de  gauche  vers  centre
Déplace the le disque supérieur de  droit  vers  centre
Déplace the le disque supérieur de  gauche  vers  droit
Déplace the le disque supérieur de  centre  vers  gauche
Déplace the le disque supérieur de  centre  vers  droit
Déplace the le disque supérieur de  gauche  vers  droit
```

Ancêtre

Avec ce programme, vous pouvez trouver l'ancêtre femme commun entre différentes personnes.

```
//Déclaration des prédicats
prédicat ancêtre,parent,homme,femme,test;

// clauses
ancêtre(?X,?X) :- vrai;
ancêtre(?X,?Z) :- parent(?X,?Y),ancêtre(?Y,?Z);

// relations familiales, enregistrées dans la base de faits
parent("george","sam") :- vrai;
parent("george","andy") :- vrai;
parent("andy","mary") :- vrai;

homme("george") :- vrai;
homme("sam") :- vrai;
homme("andy") :- vrai;

femme("mary") :- vrai;

test(?X,?Q) :- ancêtre(?X,?Q), femme(?Q);
test._trace(vrai);

//Nous explorons toutes les possibles du fait du choix d'une variable réceptricette table.
table v=test("george",?Z);
afficheligne(v);
```

Execution avec trace:

```
r:0=test(?X,?Q) --> test(george,?Z14)
e:0=test('george',?Q16) --> ancêtre('george',?Q16),femme(?Q16)
 r:1=ancêtre(?X,?X) --> ancêtre('george',?Q16),femme(?Q16)
 e:1=ancêtre('george','george') --> femme('george') --> Échec
 r:1=ancêtre(?X,?Z) --> ancêtre('george',?Q16),femme(?Q16)
 e:1=ancêtre('george',?Z19) --> parent('george',?Y20),ancêtre(?Y20,?Z19),femme(?Z19)
  k:2=parent('george','sam') --> ancêtre('sam',?Z19),femme(?Z19)
   r:3=ancêtre(?X,?X) --> ancêtre('sam',?Z19),femme(?Z19)
   e:3=ancêtre('sam','sam') --> femme('sam') --> Échec
   r:3=ancêtre(?X,?Z) --> ancêtre('sam',?Z19),femme(?Z19)
   e:3=ancêtre('sam',?Z23) --> parent('sam',?Y24),ancêtre(?Y24,?Z23),femme(?Z23)
  k:2=parent('george','andy') --> ancêtre('andy',?Z19),femme(?Z19)
   r:3=ancêtre(?X,?X) --> ancêtre('andy',?Z19),femme(?Z19)
   e:3=ancêtre('andy','andy') --> femme('andy') --> Échec
   r:3=ancêtre(?X,?Z) --> ancêtre('andy',?Z19),femme(?Z19)
   e:3=ancêtre('andy',?Z27) --> parent('andy',?Y28),ancêtre(?Y28,?Z27),femme(?Z27)
    k:4=parent('andy','mary') --> ancêtre('mary',?Z27),femme(?Z27)
     r:5=ancêtre(?X,?X) --> ancêtre('mary',?Z27),femme(?Z27)
     e:5=ancêtre('mary','mary') --> femme('mary')
      success:6=test('george','mary')
     r:5=ancêtre(?X,?Z) --> ancêtre('mary',?Z27),femme(?Z27)
     e:5=ancêtre('mary',?Z31) --> parent('mary',?Y32),ancêtre(?Y32,?Z31),femme(?Z31)
[test('george','mary')]
```

Ancêtre de nouveau mais avec une base de données

```
prédicat ancêtre,parent,homme,femme,test,truc;

sqlite db;

//Base données : ouverture ou création
db.open('persitent.db');
```

```
// déclaration des prédicats dans la base
 db.prédicat("femme",1);
 db.prédicat("parent",2);

//mode transactionnel
db.comnence();

//Enregistrement des prédicats
femme("stephanie") :- enregistre(db);
femme("liliane") :- enregistre(db);
femme("jeanne") :- enregistre(db);
femme("mary") :- enregistre(db);
femme("francoise") :- enregistre(db);

parent("george",10.3234) :- enregistre(db);
parent("george",100) :- enregistre(db);
parent("george","sam") :- enregistre(db);
parent("george","andy") :- enregistre(db);
parent("andy","mary") :- enregistre(db);

db.transmet();

//Nous indiquons au système comment accéder aux prédicats dans la base.
parent(?X,?Y) :- get(db);
femme(?X) :- get(db);

//Pas de changement avec l'exemple précédent
ancêtre(?X,?X) :- vrai;
ancêtre(?X,?Z) :- parent(?X,?Y),ancêtre(?Y,?Z);


test(?X,?Q) :- ancêtre(?X,?Q), femme(?Q);

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

afficheligne(v);

db.ferme();
```

Un exemple de traitement des langues

```
//Déclaration des prédicats
prédicat sentence,noun_phrase,det,noun,verb_phrase,verb;

//Mais aussi des termes...
term P,SN,SV,dét,nom,verbe;

sentence(?S1,?S3,P(?A,?B)) :- noun_phrase(?S1,?S2,?A), verb_phrase(?S2,?S3,?B);
noun_phrase(?S1,?S3,SN(?A,?B)) :- det(?S1,?S2,?A), noun(?S2,?S3,?B);
verb_phrase(?S1,?S3,SV(?A,?B)) :- verb(?S1,?S2,?A), noun_phrase(?S2,?S3,?B);

//Note l'utilisation du décomposeur de liste : « | »
det(["the"|?X], ?X,dét("the"))  :- vrai;
det(["a"|?X], ?X,dét("a"))  :- vrai;

noun(["cat"|?X], ?X,nom("cat"))  :- vrai;
noun(["dog"|?X], ?X,nom("dog"))  :- vrai;
noun(["bat"|?X], ?X,nom("bat"))  :- vrai;

verb(["eats"|?X], ?X,verbe("eats"))  :- vrai;

table v;

v=sentence(?X,[],?A);
afficheligne("Toutes les phrases que l'on peut générer:",v);

//we analyze a sentence
v=sentence(["the", "dog", "eats", "a", "bat"],[],?A);
afficheligne("L'analyse:",v);
```

Exécution:

Toutes les phrases que l'on peut générer:

```
[sentence(['the','cat','eats','the','cat'],[],P(SN(dét(the),nom(cat)),SV(verbe(eats),SN(dét(the),nom(cat)
))))),sentence(['the','cat','eats','the','dog'],[],P(SN(dét(the),nom(cat)),SV(verbe(eats),SN(dét(the),nom(
dog))))),sentence(['the','cat','eats','the','bat'],[],P(SN(dét(the),nom(cat)),SV(verbe(eats),SN(dét(the),n
om(bat))))),sentence(['the','cat','eats','a','cat'],[],P(SN(dét(the),nom(cat)),SV(verbe(eats),SN(dét(a),no
m(cat))))),sentence(['the','cat','eats','a','dog'],[],P(SN(dét(the),nom(cat)),SV(verbe(eats),SN(dét(a),no
m(dog))))),sentence(['the','cat','eats','a','bat'],[],P(SN(dét(the),nom(cat)),SV(verbe(eats),SN(dét(a),no
m(bat))))),sentence(['the','dog','eats','the','cat'],[],P(SN(dét(the),nom(dog)),SV(verbe(eats),SN(dét(the
),nom(cat))))),sentence(['the','dog','eats','the','dog'],[],P(SN(dét(the),nom(dog)),SV(verbe(eats),SN(dé
t(the),nom(dog))))),sentence(['the','dog','eats','the','bat'],[],P(SN(dét(the),nom(dog)),SV(verbe(eats),S
N(dét(the),nom(bat))))),sentence(['the','dog','eats','a','cat'],[],P(SN(dét(the),nom(dog)),SV(verbe(eats)
,SN(dét(a),nom(cat))))),sentence(['the','dog','eats','a','dog'],[],P(SN(dét(the),nom(dog)),SV(verbe(eats
),SN(dét(a),nom(dog))))),sentence(['the','dog','eats','a','bat'],[],P(SN(dét(the),nom(dog)),SV(verbe(eat
s),SN(dét(a),nom(bat))))),sentence(['the','bat','eats','the','cat'],[],P(SN(dét(the),nom(bat)),SV(verbe(ea
ts),SN(dét(the),nom(cat))))),sentence(['the','bat','eats','the','dog'],[],P(SN(dét(the),nom(bat)),SV(verb
e(eats),SN(dét(the),nom(dog))))),sentence(['the','bat','eats','the','bat'],[],P(SN(dét(the),nom(bat)),SV(v
erbe(eats),SN(dét(the),nom(bat))))),sentence(['the','bat','eats','a','cat'],[],P(SN(dét(the),nom(bat)),SV(
verbe(eats),SN(dét(a),nom(cat))))),sentence(['the','bat','eats','a','dog'],[],P(SN(dét(the),nom(bat)),SV(
verbe(eats),SN(dét(a),nom(dog))))),sentence(['the','bat','eats','a','bat'],[],P(SN(dét(the),nom(bat)),SV(
verbe(eats),SN(dét(a),nom(bat))))),sentence(['a','cat','eats','the','cat'],[],P(SN(dét(a),nom(cat)),SV(ver
be(eats),SN(dét(the),nom(cat))))),sentence(['a','cat','eats','the','dog'],[],P(SN(dét(a),nom(cat)),SV(verb
be(eats),SN(dét(the),nom(dog))))),sentence(['a','cat','eats','the','bat'],[],P(SN(dét(a),nom(cat)),SV(ver
be(eats),SN(dét(the),nom(bat))))),sentence(['a','cat','eats','a','cat'],[],P(SN(dét(a),nom(cat)),SV(verbe(
eats),SN(dét(a),nom(cat))))),sentence(['a','cat','eats','a','dog'],[],P(SN(dét(a),nom(cat)),SV(verbe(eats)
,SN(dét(a),nom(dog))))),sentence(['a','cat','eats','a','bat'],[],P(SN(dét(a),nom(cat)),SV(verbe(eats),SN(
dét(a),nom(bat))))),sentence(['a','dog','eats','the','cat'],[],P(SN(dét(a),nom(dog)),SV(verbe(eats),SN(d
ét(the),nom(cat))))),sentence(['a','dog','eats','the','dog'],[],P(SN(dét(a),nom(dog)),SV(verbe(eats),SN(
dét(the),nom(dog))))),sentence(['a','dog','eats','the','bat'],[],P(SN(dét(a),nom(dog)),SV(verbe(eats),SN
(dét(the),nom(bat))))),sentence(['a','dog','eats','a','cat'],[],P(SN(dét(a),nom(dog)),SV(verbe(eats),SN(d
ét(a),nom(cat))))),sentence(['a','dog','eats','a','dog'],[],P(SN(dét(a),nom(dog)),SV(verbe(eats),SN(dét(
a),nom(dog))))),sentence(['a','dog','eats','a','bat'],[],P(SN(dét(a),nom(dog)),SV(verbe(eats),SN(dét(a),
nom(bat))))),sentence(['a','bat','eats','the','cat'],[],P(SN(dét(a),nom(bat)),SV(verbe(eats),SN(dét(the),n
om(cat))))),sentence(['a','bat','eats','the','dog'],[],P(SN(dét(a),nom(bat)),SV(verbe(eats),SN(dét(the),n
om(dog))))),sentence(['a','bat','eats','the','bat'],[],P(SN(dét(a),nom(bat)),SV(verbe(eats),SN(dét(the),n
om(bat))))),sentence(['a','bat','eats','a','cat'],[],P(SN(dét(a),nom(bat)),SV(verbe(eats),SN(dét(a),nom(c
at))))),sentence(['a','bat','eats','a','dog'],[],P(SN(dét(a),nom(bat)),SV(verbe(eats),SN(dét(a),nom(dog)))
)),sentence(['a','bat','eats','a','bat'],[],P(SN(dét(a),nom(bat)),SV(verbe(eats),SN(dét(a),nom(bat)))))]
```

L'analyse:

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

Tour de Hanoï annimé

Le code suivant montre une annimation où les disques bougent de colonne en colonne sous le contrôle des prédicats.

```
//Déclaration des prédicats
prédicat déplace;

// configuration initiale... Les disques sont sur la colonne de gauche
dictionnaire colonnes={'gauche':[70,50,30],'centre':[],'droit':[]};

//nous dessinons un disque selon sa position et sa colonne
fonction disque(fenêtre w,nombre x,nombre y,nombre sz,nombre position) {
    nombre start=x+100-sz;
    nombre level=y-50*position;
    w.rectangleplein(start,level,sz*2+20,30,FL_BLEU);
}
```

```
fonction affichage(fenêtre w,omni o) {

    w.couleurdessin(FL_NOIR);
    w.police(FL_HELVETICA,40);

    w.dessinetexte("Gauche",180,200);
    w.dessinetexte("Centre",460,200);
    w.dessinetexte("Droit",760,200);

    w.rectangleplein(200,300,20,460,FL_NOIR);
    w.rectangleplein(100,740,220,20,FL_NOIR);

    w.rectangleplein(500,300,20,460,FL_NOIR);
    w.rectangleplein(400,740,220,20,FL_NOIR);

    w.rectangleplein(800,300,20,460,FL_NOIR);
    w.rectangleplein(700,740,220,20,FL_NOIR);

    //Now we draw our disques
    table gauche=colonnes['gauche'];
    table centre=colonnes['centre'];
    table droit=colonnes['droit'];
    nombre i;

    pour (i=0 tant que i<gauche faire i++)
        disque(w,100,740,gauche[i],i+1);
    pour (i=0 tant que i<centre faire i++)
        disque(w,400,740,centre[i],i+1);
    pour (i=0 tant que i<droit faire i++)
        disque(w,700,740,droit[i],i+1);

}
fenêtre w avec affichage;

//------ Partie moteur d'inférence----------
//On se déplace depuis la colonne x à la colonne y
fonction déplacement(chaine x,chaine y) {
    colonnes[y].push(colonnes[x][-1]);
    colonnes[x].pop();

    w.redessine();
    //Une petite pause avant de tout redessiner.
    pause(0.5);
    renvoie(vrai); //Important, on renvoie vrai… ou le prédicat échoue.
}


//Note the variable names, which all start avec a "?"
déplace(1,?X,?Y,_) :- déplacement(?X,?Y);

déplace(?N,?X,?Y,?Z) :-
  ?N>1,
  ?M est ?N-1,
  déplace(?M,?X,?Z,?Y),
  déplace(1,?X,?Y,_),
  déplace(?M,?Z,?Y,?X);

//L'inférence est lancée depuis une tâche...
tâche hanoi() {
    déplace(3,"gauche","droit","centre");
}
//-----------------------------------------------------------------
fonction Lance(bouton b,omni o) {
    hanoi();
}

//Il suffit de presser le bouton pour lancer l'animation
bouton b avec Lance;
w.commence(50,50,1000,800,"HANOI");
b.crée(20,20,60,30,FL_Régulier,FL_BOUTON_NORMAL,"Lance");
```

```
w.fin();
w.lance();
```

kifsys

La bibliothèque kifsys fournit un ensemble de commandes pour gérer des répertoires, ou exécuter des commandes systèmes.

Méthodes

1. commande(chaine com,chaine sortie): exécute la commande 'com'. Si « sortie » est fourni, alors l'ensemble des affichages effectués par la commande « com » est redirigé vers le fichier dont le nom est fourni par cette variable.
2. créerépertoire(chaine chemin): Crée le répertoire 'chemin'. Renvoie faux si le répertoire ne peut être créé (généralement parce qu'il existe déjà).
3. mkdir(chaine chemin): Crée le répertoire 'chemin'. Renvoie faux si le répertoire ne peut être créé (généralement parce qu'il existe déjà).
4. listerépertoire(chaine chemin): Retourne le contenu du répertoire *chemin* sous la forme d'une table de chaine.
5. ls(chaine chemin): Retourne le contenu du répertoire *chemin* sous la forme d'une table de chaine.
6. cheminabsolu(chemin vchemin): Retourne le chemin absolu
7. infosfichier(chaine chemin): Retourne les infos du fichier sous la forme d'un dictionnaire de chaine.
    a. *info["taille"]*: taille du fichier
    b. *info["date"]*: date du fichier
    c. *info["changement"]:* date du dernier changement
    d. *info["accès"]*: date du dernier accès
    e. *info["répertoire"]: vrai si le chemin est un répertoire*
    f. *info["cheminfichier"]:* le nom complet
8. env(chaine var): Retourne le contenu de la variable d'environnement.

9. env(chaine var,chaine val): Initialise la variable d'environnement *var* avec la valeur *val.*

- Exemple

```
//Cette fonction copie tous les fichiers d'un répertoire vers un nouveau s'ils sont plus récents qu'une certaine date
fonction cp(chaine lechemin,chaine tochemin) {
        //Nous lisons le contenu de le source répertoire
        table v=kifsys.listerépertoire(lechemin);

        itérateur it;
        chaine chemin;
        chaine cmd;
        dictionnaire m;
        date t;

        //La date d'aujourd'hui à partir de 9h00 du matin
        t.metdate(t.année(),t.mois(),t.jour(),9,0,0);

        it=v;
        pour (it.commence() tantque it.nfin() faire it.suivant()) {
                chemin=lechemin+'/'+it.valeur();
                //si le fichier est du bon type
                si (".cxx" dans chemin ou ".h" dans chemin ||  ".c" dans chemin) {
                        m=kifsys.infosfichiers(chemin);
                        //si le date est plus récente que notre date courante
                        si (m["date"]>t) {
                                //nous le copions
                                cmd="cp "+chemin+' '+tochemin;
                                afficheligne(cmd);
                                //Nous exécute notre commande
                                kifsys.commande(cmd);
                        }
                }
        }
}

//Nous copions nos fichiers depuis test vers sauvegarde
cp('/claude/test','/claude/sauvegarde');
```

kifsocket

kifsocket est une bibliothèque dont le but est de permettre de mettre en place des applications client/serveur.

Méthodes

1. **attends():** *attend qu'un client se connecte et renvoie son identifiant.*
2. **bloquant(bool flag)**: *Si 'flag' est 'vrai', le connecteur fonctionne en mode bloquant.*
3. **connecte(chaine hôte,nombre port):** *connexion à un serveur identifié par un nom de système hôte et un numéro de port.*
4. **créeserveur(chaine hôte,nombre port,nombre nblients):** *crée un serveur. Si l'hôte est omis, le nom local est alors utilisé.*
5. **écrit(nombre num,chaine s):** *Ecrit une chaine s sur le connecteur. Du côté serveur, 'num' est l'identifiant client renvoyé par 'attend'. En revanche du côté client, utiliser 'écrit(chaine s)'*
6. **envoie(nombre num,chaine s*):** Ecrit une chaine à destination d'une application non KIFF. 'num' doit être omis du côté client.*
7. **ferme(nombre num):** *Ferme un connecteur. Du côté serveur, si 'num' est fourni, Il ferme la connexion pour un client donné.*
8. **laclasse(chaine frame):** *Retourne un identifiant objet déclaré du côté serveur*
9. **lafonction(chaine frame):** *Retourne une fonction déclarée du côté serveur*

10. **lance(nombre num,chaine stop):** *Uniquement du côté serveur. Exécute des invocations de méthodes distantes. 'stop' est une chaine qui permet l'arrêt de ce service par le client.*
11. **lit(nombre num):** *lit une chaine sur le connecteur. Du côté serveur, num est l'identifiant du client. Utiliser 'lit()' sans paramètre du côté client.*
12. **nomclient(nombre num)**: *Retourne le nom du client*
13. **nomhôte():** *Retourne le nom de l'hôte*
14. **reçoit(nombre num)**: *Reçoit* num *caractères provenant d'une application non KIFF.*
15. **reçoit(nombre client,nombre num)**: *Reçoit* num *caractères d'un client provenant d'une application non KIFF.*
16. **saisie()**: *Reçoit* un seul *caractères provenant d'une application non KIFF.*
17. **saisie(nombre client)**: *Reçoit* un seul *caractère d'un client provenant d'une application non KIFF.*
18. **tempsmax(nombre t):** *Définit un temps maximum de 't' secondes pour le connecteur*

Exemple: côté serveur

```
//Côté serveur
nombre clientid;
connecteur s; //nous créons un connecteur
chaine nom=s.nomhôte();
afficheligne("Local serveur:",nom);
//Nous créons notre serveur pour le port 2020, avec au plus 5 connexions…
s.créeserveur(2020,5);
//nous attendons une connexion client
tantque (vrai) {
        //nous acceptons au plus cinq connexions
        clientid=s.attends();
        //nous lisons un message provenant du client
        chaine message=s.lit(clientid);
        message+=" et renvoie";
        //nous renvoyons le message au client
        s.écrit(clientid);
        //nous fermons la connexion
```

```
    s.ferme(clientid);
}
//Nous fermons le serveur
s.ferme();
```

Exemple: côté client

```
//côté client
connecteur s; //nous créons un connecteur
chaine nom=s.nomhôte();
afficheligne("Local serveur:",nom);
//Nous créons notre serveur sur le port 2020
s.connecte(nom,2020);
//nous écrivons un message au serveur
chaine message="Hello";
s.écrit(message);
//nous lisons le message en retour du serveur

message=s.lit();
afficheligne(message);
//nous fermons la connexion
s.ferme();
```

distant

kifsocket offre un second type: *distant*, qui est utilisé pour exécuter des méthodes et des fonctions sur un serveur distant. Le serveur doit lancer un *lance* pour gérer les requêtes du client.

Obtenir classes et fonctions

Le client de son côté, doit demander les classes et fonctions disponibles à travers deux méthodes de *connecteur*: *laclasse* et *lafonction*.
Par défaut, tout objet et toute fonction peut être demandé par le client, ce qui limite la nécessité de déclarations particulière du côté serveur.
Les deux méthodes renvoient un objet distant, qui peut être utilisé pour exécuter fonctions ou méthodes.

Privée

Si en revanche, vous voulez protéger des objets ou des fonctions du côté serveur, vous devez les déclarer en tant que privée.

Chaine ou table

Quand un objet distant est utilisé avec une table, Il renvoie la liste des fonctions disponibles sur le serveur. Dans le cas d'un appel à lafonction, la liste sera limitée à un seul élément. Cependant, dans le

cas de laclasse, toutes les fonctions disponibles (non privée) seront enregistrées.

Exécution

La chaine donné en paramètre à la fonction *lance* permet au client de fermer le serveur à distance, en le rappelant avec la chaine en question. Pour éviter qu'un client ne ferme malencontreusement un serveur, on peut démarrer celui-ci avec la chaine vide.

Côté serveur

Le code ci-dessous implémente une classe et une fonction. Ce serveur implémente une fonction: crée, qui crée un nouvel objet test en tant que variable globale (via un courtier d'objet), et un objet de classe toto, qui est accessible par le client.

Le serveur peut être stoppé avec une chaine specifique, ici: *stop*.

```
//Un simple serveur d'objets...

//D'abord une classe
classe montest {
        nombre i;

        //Cette fonction ne doit pas être appelée du côté client
        privée fonction _initiale(nombre x) {
                i=x;
        }

        //Cette fonction ne doit pas être appelée du côté client
        privée fonction Met(nombre j) {
                i=j;
    }

        fonction Valeur() {
                renvoie(i);
        }

        fonction Calcul(nombre j) {
                renvoie(i*j);
        }
}

//Le méthode crée un objet de classe test, avec comme nom: n
fonction crée(chaine nom,nombre i) {
        montest t(i);
       //Nous créons dans l'espace local au serveur une variable dont la valeur est t
        _KIFFPRINCIPALE [nom]=t;
}

//Nous avons notre objet, qu'un client peut réclammer
```

```
montest toto(100);

//Ce objet est declaré comme “privée” et ne peut être demandé par le client…
privée montest pastouche(-1);

//Nous créons notre serveur
connecteur s;

afficheligne(s.nomhôte());
//Port: 2012
s.créeserveur(2012,10);

//Attente de connexion
nombre client=s.attends();
//Chaine d’arrêt: "stop"
//Nous sommes prêt à exécuter des requêtes de la part des clients...
s.lance(client,"stop");
//Si notre serveur reçoit le "stop", Il s’arrête.
s.ferme();
afficheligne("Stoppée");
```

Côté client

Le client implémente un appel à l’objet toto. Il appelle aussi *crée* pour créer un nouvel objet du côté serveur. Finalement, Il écrit la chaine stop qui tue le serveur.

```
//Nous nous connectons à notre serveur.
connecteur s;

chaine nom=s.nomhôte();
s.connecte(nom,2012);

//Notre objet distant: r
distant r;

//Nous récupérons un pointeur sur l’objet toto sur le serveur
r=s.laclasse("toto");

//Nous affichons les méthodes disponibles de la description de la classe de toto, qui a été renvoyé par le serveur.
afficheligne("Fonctions:",r);

afficheligne("Valeurs de toto:",r.Valeur(),r.Calcul(345));

//Nous allons maintenant récupérer une fonction: crée
r=s.lafonction("crée");

//Ce qui nous permet de créer un objet titi du côté serveur initialisé avec la valeur 123...
r.crée("titi",123);

//nous pouvons aller chercher maintenant l’objet titi désormais disponible sur le serveur
r=s.laclasse("titi");
```

```
// nous affichons ses valeurs
afficheligne("Valeur de titi:",r.Valeur(),r.Calcul(345));

//Nous tuons le serveur.
s.écrit("stop");
s.ferme();
```

kifsqlite

KIFF fournit aussi une bibliothèque pour gérer une base de données de type sqlite.

Méthodes

1. **commence():** *Entre dans un mode différé*
2. **crée(x1,x2,x3):** *crée une table dans la base de données avec les arguments x1,x2,x3. Ex. mydb.crée('table1','nom TEXT PRIMARY KEY','age INTEGER');*
3. **lance(chaine sqlcommand):** *Exécute une commande SQL. Si la variable de réception est une table, toutes les valeurs extraites seront placées dedans. Si la variable de réception est un itérateur, alors chaque valeur sera un dictionnaire dont les noms seront les attributs de la table.*
4. **exécute(chaine sqlcommand) :** *Exécute une commande SQL, sans valeur de retour.*
5. **ferme():** *ferme une base de données*
6. **insère(chaine table,chaine colonne,nombre valeur,...):** *insère une ligne dans une table. Ex. mydb.insère('table1','nom',nm,'age',i);*
7. **ouvre(chaine chemin):** *ouvre un fichier de base de données*
8. **transmet():** *Fin du mode différé. L'ensemble des commandes est alors transmis à la base de données pour exécution.*

Exemple

```
//nous déclarons une nouvelle variable sqlite
sqlite mydb;

//nous ouvrons une base de données
mydb.ouvre('test.db');

tente {
        //nous insérons une nouvella table dans la base de données
 mydb.crée("table1","nom TEXT PRIMARY KEY","age NUMBER");
```

```
 afficheligne("table1 est créee");
}
capte() {
        //Cette base existe déjà
        afficheligne("Déjà créee");
}

nombre i;
chaine nm;
//Nous insérons des valeurs dans la base en mode différé, beaucoup plus rapide
mydb.commence();
//Nous insérons 5000 éléments
for(i=0;i<5000;i+=1) {
        nm="tiia_"+i;
        tente {
                //nous insérons dans table1 deux valeurs, pour 'nom' et 'age'.
                //Notez l'alternance entre le nom des colonnes et leurs valeurs.
                mydb.insert("table1","nom",nm,"age",i);
                afficheligne(i);
        }
        capte() {
                afficheligne("Déjà insérée");
        }
}
//nous pouvons enfin transmettre l'ordre de traitement des insertions.
mydb.transmet();

//nous itérons parmi notre valeurs pour une commande SQL donnée
itérateur it=mydb.lance("select * from table1 where age>10;");
pour (it.commence() tantque it.nfin() faire it.suivant())
   afficheligne("Valeur: ",it.valeur());

//Nous aurions pu obtenir le même résultat avec
//table v=mydb.exécute("select * de table1 où age>10;");
//Cependant le risque de débordement dans notre table rend l'opération risquée

mydb.ferme();
```

Bibliothèque Fast Light ToolKit (GUI)

FLTK (http://www.fltk.org/) est une biblitohèque graphique C++, qui a été implémentée pour de nombreuses plate-formes, de Windows à Mac Os. Nous avons encapsulé FLTK en une bibliothèque KIFF, de façon à enrichir le langage avec des capacités graphiques. La plupart des méthodes de FLTK ont été portées.

Note: Nous utilisons pour l'instant la version 1.3.0 de FLTK.

Méthodes communes

La plupart des objets décrit dans ce document partage un certain nombre de méthodes communes :

Ces méthodes lorsqu'elles sont appelées sans paramètre renvoient leur valeur courante correspondante.

Méthodes

1. aligne(nombre): définit l'alignement du label (voir plus bas)
2. couleurarrièreplan(nombre c|chaine c): initialise la couleur d'arrière-plan
3. coords(nombre c): renvoie la table des coordonnées du widget
4. coords(nombre x,nombre y,nombre w,nombre h): définit les coordonnées du widget.
5. créée(): renvoie vrai si l'objet a été correctement créée
6. cache(): Cache le widget
7. label(): Renvoie le label du noeud.
8. couleurlabel(nombre c): définit ou renvoie la couleur de la police du label
9. policelabel(nombre c): définit ou renvoie la police du label
10. taillelabel(nombre i): définit ou renvoie la taiite de la police du label
11. typelabel(nombre i): définit ou renvoie le type du label (voir ci-dessous pour une description des differents types)
12. couleursélection(nombre couleur): Couleur de l'élément sélectionné
13. montre(): montre le widget

14. truc(chaine txt): associe un truc à un widget. Un *truc* est une explication du widget qui s'affiche lorsque l'on passe le curseur dessus.

Type de label

FL_LABEL_NORMAL
FL_PAS_DE_LABEL
FL_LABEL_OMBRE
FL_LABEL_GRAVE
FL_LABEL_BOSSELE

Alignement

FL_LIGNE_CENTRE
FL_LIGNE_HAUT
FL_LIGNE_BAS
FL_LIGNE_GAUCHE
FL_LIGNE_DROIT
FL_LIGNE_DEDANS
FL_LIGNE_TEXTE_DESSUS_IMAGE
FL_LIGNE_IMAGE_DESSUS_TEXTE
FL_LIGNE_CLIP
FL_LIGNE_ENROULE
FL_LIGNE_IMAGE_ACOTE_TEXTE
FL_LIGNE_TEXTE_ACOTE_IMAGE
FL_LIGNE_IMAGE_BACKDROP
FL_LIGNE_HAUT_GAUCHE
FL_LIGNE_HAUT_DROIT
FL_LIGNE_BAS_GAUCHE
FL_LIGNE_BAS_DROIT
FL_LIGNE_GAUCHE_HAUT
FL_LIGNE_DROIT_HAUT
FL_LIGNE_GAUCHE_BAS
FL_LIGNE_DROIT_BAS
FL_LIGNE_NOWRAP
FL_LIGNE_POSITION_MASQUE
FL_LIGNE_IMAGE_MASQUE

bitmap

Cet type d'objet permet de construire à partir d'une table d'entiers un bitmap. Un bitmap est une image dans laquelle chaque bit dans la table d'entiers correspond à un pixel.

Méthode

charge(tablenombres btm,nombre dimX,nombre dimY): charge un bitmap dont les dimensions sont données.

Exemple

```
//Dessine une sorte de B...
tablenombres iv=[254,63,60,28,24,8,8,24,8,24,8,8,204,15,
204,31,12,48,8,96,8,96,8,96,8,32,12,56,252,31,31,0];

nombre dx=16,dy=16;

bitmap b;

b.charge(iv,dx,dy);

fonction affiche(fenêtre f, omni e) {
    f.bitmap(b,FL_ROUGE,50,50);
}

fenêtre f avec affiche;

f.crée(30,30,500,500,"Une sorte de B");
f.lance();
```

image

Ce objet est utilisé pour charger une image de type GIF ou JPEG, qui peut désormais être utilisé avec un objet fenêtre ou un objet bouton, à travers la méthode *image*.

Méthodes

1. chargejpeg(chaine fichiernom): charge une image JPEG
2. chargegif(chaine fichiernom): charge une image GIF

fenêtre

Le type fenêtre est le type de base. Il comprend un grand nombre de méthodes qui permettent l'affichage et l'habillage des boutons, glissières et autres widgets.

Méthodes

1. àlafermeture(fonction,objet): Fonction de rappel lors de la fermeture
2. alerte(chaine msg): fenêtre pour afficher une alerte
3. aligne(nombre): définit l'alignement du label
4. arc(nombre x,nombre y,nombre x1, nombre y1, décimal a1, décimal a2): Dessine un arc.
5. arc(décimal x,décimal y,décimal rad,décimal a1,décimal a2): Ajoute une série de points en arc de cercle au chemin courant.

6. autemps(fonction,t,objet): Initialise une fonction de rappel pour chaque intervalle de temps t
7. bitmap(bitmap image,nombre couleur,nombre x, nombre y): Affiche un bitmap avec une couleur particulière à la position x,y.
8. bitmap(bitmap image,nombre couleur,nombre x, nombre y, nombre w, nombre h): Affiche un bitmap avec une couleur particulière. Construit le bitmap dans une boite dont les dimensions sont fournies.
9. bords(bool b): Si vrai ajoute ou retire les bords. Si pas de paramètres renvoie si la fenêtre a des bords
10. boucle(nombre x,nombre y,nombre x1, nombre y1,nombre x2, nombre y2, nombre x3, nombre y3): Dessine une série de lignes, x3 et y3 sont optionnels
11. cache(bool h): Cache la fenêtre si hauteur est vrai
12. camembert(nombre x,nombre y,nombre x1, nombre y1, décimal a1, décimal a2): Dessine un camembert
13. cercle(nombre x,nombre y,nombre r,nombre couleur): Dessine un cercle. 'couleur' est optionnelle.
14. chasse(): force à redessiner la fenêtre si celle-ci est dans une tâche
15. colle(): Colle chaine depuis le presse-papier
16. commence(nombre x,nombre y,nombre largeur, nombre hauteur,chaine titre): Crée une fenêtre et commence l'initialisation, largeur et hauteur sont optionnels
17. coords(nombre c): renvoie la table des coordonnées du widget
18. copie(chaine): Copie la chaine dans le presse-papier
19. couleurarrièreplan(nombre c|chaine c): initialise la couleur d'arrière-plan
20. couleurdessin(nombre c|chaine c): Couleur des prochains objets dessinés
21. couleurlabel(nombre c): définit ou renvoie la couleur du label
22. couleurrvb(chaine|nombre couleur)|(nombre r,nombre v,nombre b)|(table couleur): Renvoie soit une table de la décomposition en couleur ou renvoie la valeur de la couleur correspondant à la combinaison RVB fournie
23. couleursélection(nombre couleur): Couleur de l'élément sélectionné
24. courbe(décimal x,décimal y,décimal x1,décimal y1,décimal x1,décimal y2,décimal x3,décimal y3): Ajoute une série de

points sur une courbe de Bezier au chemin courant. La fin de la courbe (et deux de ses points) est à la position x,y et x3,y3.

25. crée(nombre x,nombre y,nombre largeur, nombre hauteur,chaine titre): Crée une fenêtre sans widgets, largeur et hauteur sont optionnels
26. créée(): Renvoie vrai si l'objet a été crée
27. demande(chaine msg,chaine boutonmsg1,chaine boutonmsg2,chaine boutonmsg3): Fenêtre pour poser une question
28. dessinepoints(tabledécimaux xy,int épaisseur,tabledécimaux repère): Dessine un graphe à partir des coordonnées successives x,y dans xy. Cette méthode calcule automatiquement en fonction de l'aire de la fenêtre les dimensions idéales pour que tous les points apparaissent à l'écran. *Epaisseur* définit la taille des points. Si épaisseur==0, alors on joint tous les points par des lignes. Elle renvoie une table de décimaux correspondants aux éléments nécessaires pour calculer une nouvelle position avec *coordsdessinepoints*. Cette table a le format suivant : [*minXFenêtre, minYFenêtre, maxXFenêtre, maxYFenêtre, minXValeur, minYValeur, maxXValeur,maxYValeur,incrémentX, incrémentY, épaisseur*]. *Repère* est une table optionnelle qui a exactement la même forme que ci-dessus, mais où l'on peut omettre les minXValeur...maxYValeur ainsi que les incréments. Cette table a donc les tailles possibles suivantes: 4,8,10 ou 11.
29. coordsdessinepoints(tabledécimaux repère,décimal x, décimal y): renvoie les coordonnées fenêtre [xe,ye] d'un point x,y en fonction des paramètres d'échelle calculés pour dessiner le graphe avec *dessinepoints*. *repère* est la table renvoyé par *dessinepoints.*
30. dessinetexte(chaine l,nombre x,nombre y): Place un texte à la position x,y
31. débutboucle(): Commence à dessiner des séquences de lignes qui se referment.
32. débutligne(): Commence à dessiner des lignes.
33. débutpoints(): Commence à accumuler des sommets
34. débutpolygone(): Commence à dessiner un polygone plein convexe
35. débutpolygonecomplexe(): Commence à dessiner un polygone complexe

36. dépileclip(): Relache une région clippée
37. dépilematrice(): Restaure la transformation courante
38. déverrouille(): Déverrouillage tâche
39. échelle(décimal x,décimal y)|(x): Met à l'échelle la transformation courante.
40. empileclip(nombre x,nombre y,nombre largeur, nombre h): Définit une région clippée de coordonnées x,y,largeur,h
41. empilematrice(): Sauvegarde la transformation courante
42. ferme(): ferme la fenêtre
43. fin(): fin création
44. finboucle(): Cesse d'accumuler des séquences de lignes qui se referment.
45. finligne(): Cesse d'ajouter des lignes
46. finpoints(): Cesse d'accumuler des sommets
47. finpolygone(): Cesse de dessiner un polygone plein convexe
48. finpolygonecomplexe(): Cesser de dessiner un polygone complexe
49. focus(): Récupère le focus
50. formeligne(chaine type,nombre largeur): Sélectionne la forme de la ligne et son épaisseur
51. image(image image,nombre x, nombre y, nombre largeur, nombre h): Affiche une image
52. initialisepolices(): Charge les polices systèmes. Renvoie le nombre de polices disponibles
53. label(chaine s): définit ou renvoie le texte du label
54. lance(): lance la boucle principale de l'interface graphique
55. saisie(chaine msg) : lance une fenêtre pour permettre la saisie d'une valeur.
56. ligne(nombre x,nombre y,nombre x1, nombre y1,nombre x2, nombre y2): Dessine une ligne entre deux points, x2 et y2 sont optionnels
57. menu(table,nombre x,nombre y,nombre largeur, nombre h): initialise un menu avec ses fonction de rappels
58. modal(bool b): Si vrai rfin la fenêtre modale. Si pas de paramètres renvoie si la fenêtre est modale
59. montre(): Montre la fenêtre
60. motdepasse(chaine msg) : affiche une fenêtre pour saisir un mot de passe.
61. multmatrice(décimal a,décimal b,décimal c,décimal d, décimal x, décimal y): combine les transformations

62. nompolice(nombre num): nom de la police.
63. nombrepolice(): Renvoie le nombre de polices disponibles.
64. pivote(décimal d): tourne de d degrés la transformation courante
65. point(nombre x,nombre y): Dessine un pixel
66. police(chaine f,nombre sz): Définit le nom de la police et sa taille
67. policelabel(nombre c): définit ou renvoie la police du label
68. polygone(nombre x,nombre y,nombre x1, nombre y1,nombre x2, nombre y2, nombre x3, nombre y3): Dessine un polygone, x3 et y3 sont optionnels
69. position()|(nombre x,nombre y): Renvoie la position de la fenêtre ou définit la position de la fenêtre
70. rectangle(nombre x,nombre y,nombre largeur, nombre hauteur, chaine c|nombre c): Dessine un rectangle avec la couleur optionnelle c
71. rectangleplein(nombre x,nombre y,nombre largeur, nombre hauteur, chaine c|nombre c): Remplit un rectangle avec la couleur optionnelle c
72. redessine(): Redessine la fenêtre
73. redimensionnable(objet): objet redimensionnable
74. réveil(): Réveille une fenêtre définie dans une tâche
75. siglissedépose(fonction,objet): Initialise la fonction de rappel lors d'un glisser-déposer
76. sisouris(nombre action, fonction,objet): Définit le nom de la fonction de rappel pour une action souris
77. sitouche(nombre action, fonction,objet): Définit le nom de la fonction de rappel pour une action clavier
78. sommet(décimal x,décimal y): Ajoute un sommet à une structure complexe
79. stylecurseur(nombre curseurtype, nombre couleur,nombre couleur): Définit la forme du curseur
80. taille()|(nombre x,nombre y,nombre largeur, nombre h): Renvoie la taille de la fenêtre ou définit la taille de la fenêtre
81. taillelabel(nombre c): définit ou renvoie la taille de la police du label
82. taillemax(nombre minw,nombre minh, nombre maxw,nombre maxh): définit les tailles maximum d'agrandissement de la fenêtre
83. taillepolices(nombre num): renvoie une table des tailles disponibles de police.

84. tailletexte(chaine l): Renvoie un dictionnaire avec largeur et hauteur comme clef pour donner les largeurs et hauteurs de la chaine en pixels
85. transformedx(décimal x,décimal y): Transforme la distance DX sur la base de la matrice courante de transformation
86. transformedy(décimal x,décimal y): Transforme la distance DY sur la base de la matrice courante de transformation
87. transformesommets(décimal x,décimal y): ajoute les transformations à la liste des sommets
88. transformex(décimal x,décimal y): Transforme les coordonnées en X sur la base de la matrice courante de transformation
89. transformey(décimal x,décimal y): Transforme les coordonnées en Y sur la base de la matrice courante de transformation
90. translate(décimal x,décimal y): translate la transformation courante
91. trou(): remplit les trous lors du dessin d'un polygone complexe.
92. typelabel(nombre c): définit ou renvoie le type du label
93. verrouille(): Verrouille tâche

àlafermeture

Il est possible d'intercepter la fermeture d'une fenêtre via une fonction de rappel. Cette fonction de rappel doit être fourni à la fenêtre via un appel à la méthode : *àlafermeture*.

La fonction de rappel doit avoir la forme suivante:

fonction fermeture(fenêtre w,monobjet o);

Si cette fonction renvoie faux alors la fermeture de la fonction n'est pas prise en compte.

Exemple

```
fonction fermeture(fenêtre w, bool fermée) {
	si (fermée==faux) {
		afficheligne("Nous ne pouvons fermer cette fenêtre");
		renvoie(faux);
	}
	renvoie(vrai);
}

//Nous déclarons d'abord notre fenêtre.
fenêtre w;
bool fermée=faux;
//Nous créons notre fenêtre
```

```
w.crée(300,200,1300,150,"Modification");
w.àlaferemeture(fermeture,fermée);
```

Exemple de dessin d'une courbe

```
tabledécimaux fxy;


//On calcule d'abord nos coordonnées, que l'on range dans fxy, x puis y.
fonction mespoints() {
    décimal x,y;
    pour (x dans <-20,20,0.1>) {
        y=x*x*x-10;
        fxy.empile(x);
        fxy.empile(y);
    }
}

//La fonction de dessin
fonction graphe(fenêtre w,omni o) {
    w.couleurdessin(FL_NOIR);
    //On dessine notre courbe et on renvoie les informations nécessaires pour
    //effectuer les calculs d'échelle. En particulier, [maxXFenêtre,maxYFenêtre,
   //minX,minY,maxX,maxY,incrémentX,incrémentY]
    tabledécimaux repère=w.dessinepoints(fxy,0);

    //On calcule les coordonnées écran de la position (0,0)
    tabledécimaux axes=w.coordsdessinepoints(repère,0,0);

    //On dessine nos axes...
    w.ligne(axes[0],0,axes[0],repère[3]);
    w.ligne(0,axes[1],repère[2],axes[1]);
}


mespoints();
fenêtre w avec graphe;

w.commence(30,30,1000,800,"Graphe");
w.taillemax(30,30,2000,2000);
w.fin();
 w.lance();
```

autemps

Il est possible de définir une fonction qui est appelé chaque t^{ième} de seconde. Cette fonction doit avoir les paramètres suivants:

```
fonction autemps_rappel(fenêtre w, objet o);
```

Important:

Si Cette fonction renvoie 0, alors l'horloge est stoppée. En revanche, si cette fonction renvoie une valeur *décimal* quelconque, celle-ci sert à réinitialiser l'horloge pour une nouvelle interception.

Exemple

```
// fonction de rappel
fonction temps(fenêtre w,omni n) {
        afficheligne("Ok");
        renvoie(0.1);
}

fenêtre w;

w.commence(40,40,400,500,"Affichage");
w.autemps(temps,0.1,nulle);
w.fin();
w.lance();
```

Couleurs

Kifltk implémente une façon relativement simple de jouer avec les couleurs.

Tout d'abord, les couleurs prédéfinies sont les suivantes :

FL_COULEUR_AVANTPLAN
FL_COULEUR_ARRIEREPLAN2
FL_COULEUR_ARRIEREPLAN
COULEUR_INACTIVE
FL_COULEUR_SELECTION
FL_GRI0
FL_FONCE3
FL_FONCE2
FL_FONCE1
FL_LEGER1
FL_LEGER2
FL_LEGER3
FL_NOIR
FL_ROUGE
FL_VERT
FL_JAUNE
FL_BLEU
FL_MAGENTA
FL_CYAN
FL_ROUGE_FONCE
FL_VERT_FONCE
FL_JAUNE_FONCE
FL_BLEU_FONCE
FL_MAGENTA_FONCE

FL_CYAN_FONCE
FL_BLANC

Il est aussi possible de définir vos propres couleurs, grâce à un encodage RVB (Rouge Vert Bleu). Il s'agit de fournir au système une combinaison de ces trois couleurs pour générer celle voulue et ceci grâce à la méthode : *couleurrvb.* Il existe trois façons de l'utiliser :

a) table rgb= ***couleurrvb***(nombre c): cette méthode renvoie une table contenant la décomposition de la couleur c en ses composantes RVB.
b) nombre c= ***couleurrvb*** (table rvb): cette méthode prend en entrée une table de trois éléments, correspondant dans l'ordre au taux de rouge, de vert et de bleu.
c) nombre c= ***couleurrvb*** (nombre r,nombre v,nombre b): comme ci-dessus, mais les valeurs sont données individuellement.

Chaque composante est une valeur comprise entre 0 et 255…

Police

Kiflitk fournit les polices de base suivantes:

FL_HELVETICA
FL_HELVETICA_GRAS
FL_HELVETICA_ITALIQUE
FL_HELVETICA_GRAS_ITALIQUE
FL_COURIER
FL_COURIER_GRAS
FL_COURIER_ITALIQUE
FL_COURIER_GRAS_ITALIQUE
FL_TIMES
FL_TIMES_GRAS
FL_TIMES_ITALIQUE
FL_TIMES_GRAS_ITALIQUE
FL_SYMBOLE
FL_ECRAN
FL_ECRAN_GRAS
FL_ZAPF_DINGBATS
FL_FREE_FONT
FL_GRAS
FL_ITALIQUE
FL_GRAS_ITALIQUE

Exemple

Il est aussi possible comme le montre l'exemple ci-dessous, de récupérer les polices existantes sur votre machine.

```
fenêtre w;
dictionnaire styles;
éditeur wo;
nombre ipolice=0;

//nous modifions la feuille de style de notre éditeur pour voir l'usage de ces polices
fonction choixpolice(nombre idfont) {
        //nous style par défaut
        styles["#"]=[FL_NOIR,idfont,16];
        wo.ajoutestyle(styles);
        //nous affichons la police et sa taille dans le bandeau de l'éditeur
        wo.label(w.nompolice(idfont)+":"+idfont);
        //nous redessinons la fenê
        w.redessine();
}


//Quand le bouton "suivant" est pressé nous changeons notre police courante
fonction change(bouton b,nombre i) {
        choixpolice(ifont);
        ifont++;
}



bouton b(ifont) avec change;

w.commence(50,50,800,500,"Font Affichage");
w.taillemax(10,10,0,0);


nombre i;
//Nous récupérons les polices disponibles sur le système
nombre nb=w.initialisepolices();

//Nous créons notre éditeur
wo.crée(70,30,730,460,"Polices");
//Style par défaut avec une police de base
styles["#"]=[FL_NOIR,FL_HELVETICA,16];
wo.ajoutestyle(styles);

//Puis nous bouclons parmi toutes les polices disponibles pour récupérer leur nom et leur taille.
[0] means that every taille est disponible
chaine s,fonts;
table v;
pour (i=0;i<nb;i++) {
        //On met un retour chariot une fois la première police trouvée
        si (fonts!="")
                fonts+="\r";
        s=w.nompolice(i);
        v=w.taillepolice(i);
        fonts+=i+":"+s+"="+v;
}

//nous enregistrons ces noms comme contenu de l'éditeur
wo.valeur(fonts);

//Le bouton suivant
b.crée(10,10,40,30,FL_Régulier,FL_BOUTON_NORMAL,"Suivant");
w.fin();
w.redimensionnable(wo);
```

```
w.lance();
```

Type alignement

FL_ALIGNE_CENTRE
FL_ALIGNE_HAUT
FL_ALIGNE_BAS
FL_ALIGNE_GAUCHE
FL_ALIGNE_DROIT
FL_ALIGNE_DEDANS
FL_ALIGNE_TEXTE_DESSUS_IMAGE
FL_ALIGNE_IMAGE_DESSUS_TEXTE
FL_ALIGNE_CLIP
FL_ALIGNE_ENROULE
FL_ALIGNE_IMAGE_ACOTE_TEXTE
FL_ALIGNE_TEXTE_ACOTE_IMAGE
FL_ALIGNE_IMAGE_BACKDROP
FL_ALIGNE_HAUT_GAUCHE
FL_ALIGNE_HAUT_DROIT
FL_ALIGNE_BAS_GAUCHE
FL_ALIGNE_BAS_DROIT
FL_ALIGNE_GAUCHE_HAUT
FL_ALIGNE_DROIT_HAUT
FL_ALIGNE_GAUCHE_BAS
FL_ALIGNE_DROIT_BAS
FL_ALIGNE_NOWRAP
FL_ALIGNE_POSITION_MASQUE
FL_ALIGNE_IMAGE_MASQUE

Forme des lignes

Kifltk fournit toute une série de forme de lignes:

FL_SOLIDE
FL_TIRET
FL_POINTILLE
FL_POINTTIRET
FL_TIRETPOINTPOINT
FL_CAP_PLAT
FL_CAP_ROND
FL_CAP_CARRE
FL_JOINT_MITER
FL_JOINT_ROND
FL_JOINT_BEVEL

Forme des curseurs

FL_CURSEUR_DEFAUT
FL_CURSEUR_FLECHE
FL_CURSEUR_CROIX
FL_CURSEUR_SABLIER
FL_CURSEUR_INSERE
FL_CURSEUR_MAIN

FL_CURSEUR_AIDE
FL_CURSEUR_BOUGE
FL_CURSEUR_NS : flèche haut/bas (Nord/Sud)
FL_CURSEUR_OE : flèche gauche/droite (Ouest/Est)
FL_CURSEUR_NOSE : flèche diagonale
FL_CURSEUR_NESO : flèche diagonale
FL_CURSEUR_AUCUN
FL_CURSEUR_N
FL_CURSEUR_NE
FL_CURSEUR_E
FL_CURSEUR_SE
FL_CURSEUR_S
FL_CURSEUR_SO
FL_CURSEUR_O
FL_CURSEUR_NO
FL_CURSEUR_NORMAL
FL_CURSEUR_CHARIOT
FL_CURSEUR_DIM
FL_CURSEUR_BLOC
FL_CURSEUR_LOURD
FL_CURSEUR_LIGNE

Ma première fenêtre

Construire une interface graphique dans FLTK est fort simple. On ouvre d'abord une fenêtre après l'avoir déclarée, avec la méthode *commence*. Puis l'on crée tous les objets graphiques que le désire intégrer et enfin on appelle *fin.*

Une fois que la fenêtre a engrangé tous ses *widgets* ou objets graphiques, on lance la boucle qui permettra d'intercepter tous les événements clavier ou souris par rapport à cette fenêtre.

Il faut noter que *seule la fenêtre principale peut lancer « lance ».* Vous pouvez créer d'autres fenêtres de la même façon, mais il ne peut exister qu'une seule boucle d'événements et donc un seul *lance*.

```
//Déclaration de notre fenêtre
fenêtre w;
//Début de l'instanciation
w.commence(300,200,1300,150,"Modification");
//Nous définissons les bornes de notre fenêtre
w.taillemax(10,20,0,0);
//nous créons une entrée, qui sera placée dans la fenêtre
txt.crée(200,20,1000,50,vrai,"Selection");
//Plus d'autre objet, nous terminons la construction de notre fenêtre
w.fin();

//lancement de la boucle d'événement pour cette fenêtre principale
w.lance();
```

Si nous ne voulons pas ajouter de *widgets* dans la fenêtre, nous pouvons appeler *crée* à la place de *commence* et de *fin.*

Dessiner dans une fenêtre

Si vous désirez dessiner des choses dans votre fenêtre, il vous faut définir une fonction de rappel qui vous associerez avec votre fenêtre avec l'opérateur *avec.*

fenêtre wnd(objet) avec rappel_fenêtre;

Tout d'abord, *rappel_fenêtre* est la fonction qui sera appelé pour dessiner votre fenêtre chaque fois qu'un événement surviendra. Deuxièmement, *objet* dans *wnd(objet)* est un objet que vous pouvez passer à cette fonction de rappel pour effectuer des calculs ou des mise-à-jour particulières.

La fonction de rappel présente la forme suivante :

fonction rappel_fenêtre(fenêtre w, omni o) {…}

w est notre fenêtre courante, tandis que *o* est l'objet qui a été déclaré plus haut lors de la déclaration de votre objet fenêtre.

Vous pouvez dès lors enchainer des instructions graphiques comme le montre l'exemple ci-dessous :

Exemple

```
//Une classe pour garder nos coordonnées
classe mycoord {

        nombre couleur;
        nombre x,y;

        fonction _initiale() {
                couleur=FL_ROUGE;
                x=10;
                y=10;
        }
}

//nous déclarons notre objet
mycoord coords;

// affichage...
//Chaque fois que la fenêtre sera modifiée, Cette fonction sera appelée avec un objet mycoord
fonction affichage(fenêtre w,mycoord o) {
        //nous choisissons notre couleur
        w.couleurdessin(o.couleur);
        //Une forme différente de ligne
```

```
        w.formeligne(FL_POINTILLE,10);
        //nous dessinons un rectangle
        w.rectangle(o.x,o.y,250,250);
        //avec du texte...
        w.dessinetexte("TEST",100,100);
}

//nous déclarons notre fenêtre avec son objet et sa fonction de rappel
fenêtre wnd(coords) avec affichage;

//Nul besoin de widgets…
wnd.crée(100,100,300,300,"Dessin");
wnd.lance();
```

Souris

Il est aussi possible de traquer les actions de la souris via une fonction de rappel. Il faut dans ce cas utiliser la méthode *sisouris* pour associer les actions de notre souris à une fonction de rappel.

sisouris(action,rappel,monobjet);

1) action doit avoir une des valeurs suivantes:

FL_POUSSE: quand un bouton de la souris a été poussé
FL_RELACHE: quand un bouton de la souris a été relâché
FL_BOUGE: quand la souris bouge
FL_DRAGUE: quand la souris « drague » un objet
FL_ROUESOURIS: quand la roue est tournée

2) La fonction de rappel doit avoir la signature suivante:

fonction rappel_souris(fenêtre w, dnombres coords, type myobjet);

Le premier paramètre est la fenêtre elle-même. Le second paramètre est un dnombres avec les clefs suivantes:

coords[“bouton”]	la valeur du dernier bouton pressé (1,2 ou 3)
coords[“x”]	la coordonnée X dans la fenêtre de la souris
coords[“y”]	la coordonnée Y dans la fenêtre de la souris
coords[“xracine”]	la coordonnée absolue en X de la souris
coords[“yracine”]	la coordonnée absolue en Y de la souris
coords[“rouex”]	l’incrément sur l’axe X de la roue
coords[“rouey”]	l’incrément sur l’axe Y de la roue

3) monobjet est l’objet qui sera passé à la fonction de rappel

Exemple

```
//nous déclarons notre objet
mycoord coords;
```

```
// affichage...
//Chaque fois que la fenêtre sera modifiée, Cette fonction sera appelé avec un objet
mycoord
fonction affichage(fenêtre w,mycoord o) {
        //nous choisissons notre couleur
        w.couleurdessin(o.couleur);
        //a different line shape
        w.formeligne(FL_POINTILLE,10);
        //nous dessinons un rectangle
        w.rectangle(o.x,o.y,250,250);
        //avec du texte...
        w.dessinetexte("TEST",100,100);
}


//Cette fonction sera appelé pour chaque mouvement de la souris
//o est le même objet que celui associé à la fenêtre

fonction déplace(fenêtre w,dnombres souriscoord,mycoord o) {
        //nous définissons les coordonnées du rectangle dans la fonction
        //ci-dessus
        o.x=souriscoord["x"];
        o.y=souriscoord["y"];
        //nous redessinons notre fenêtre...
        w.redessine();
}

//déclaration de la fenêtre avec coords et la fonction de rappel : affichage
fenêtre wnd(coords) avec affichage;

//Début de la construction de notre fenêtre
wnd.commence(100,100,300,300,"Rectangle");
wnd.taillemax(10,10,0,0);
//Instanciation de la fonction de rappel de notre souris,
//avec l'action FL_BOUGE. Dès que la souris bouge, on redessinera notre
//rectangle
wnd.sisouris(FL_BOUGE,déplace,coords);
//la fin...
wnd.fin();

wnd.lance();
```

Clavier

Il est aussi possible d'associer une touche avec une fonction de rappel grâce à la méthode :

sitouche(action,rappel,monobjet);

1) action doit avoir l'une des valeurs suivantes:

   FL_TOUCHEPRESSEE: quand la touche est en bas
   FL_TOUCHERELACHEE: quand la touche est en haut

2) La fonction de rappel doit avoir la signature suivante:

fonction rappel_touche(fenêtre w, chaine sclef,nombre iclef, nombre combinaison, monobjet objet);

Le premier paramètre est la fenêtre elle-même. Le second paramètre est la chaine correspondant à la touche pressée, le troisième paramètre est le code numérique de cette touche. Le quatrième paramètre est la combinaison des touches de fonction pressées avec la touche courante.

3) l'objet qui a été fourni avec la fonction de rappel.

Combinaison

Il s'agit d'un entier dont les bits sont codés de la façon suivante:

- 1=la touche *ctrl* a été pressée
- 2= la touche *alt* a été pressée
- 4= la touche *commande* a été pressée
- 8=la touche *maj* a été pressée

Exemple

```
fonction poussée(fenêtre w,chaine sclef,nombre iclef,nombre comb,mycoord o) {
//Si la touche pressée est "J", alors nous bougeons notre rectangle de 10 pixels le long de la diagonale.
	si (sclef=="J") {
		o.x+=10;
		o.y+=10;
		//nous redessinons notre fenêtre
		//pour forcer le rappel de affichage
		w.redessine();
	}
}


fenêtre wnd(coords) avec affichage;

wnd.commence(100,100,300,300,"Dessin");
//la fenêtre sera redimensionnable
wnd.taillemax(10,10,0,0);
//nous ajoutons un rappel chaque fois qu'une touche est pressée…
wnd.sitouche(FL_TOUCHEPRESSEE,poussée,coords);
//le fin...
wnd.fin();
wnd.lance();
```

Comment ajouter un menu

Ajouter un menu requiert un peu plus de travail. Un menu est constitué d'une série de menus supérieurs, associés avec des sous-menus, eux-mêmes associés avec des fonctions de rappel.

fonction rappel_menu(fenêtre w,monobj obj);

où obj est un objet fourni par l'utilisateur.

Un menu est décrit à travers une table, où le premier élément est le nom du menu, suivi par une série d'autres tables, où chaque élément est un sous-menu.

```
table menu;
menu.empile(["&Fichier",["&Nouveau
Fichier",[FL_COMMANDE,"o"],cmenu1,obj,vrai],
                        ["&Ouvrir Fichier",[FL_COMMANDE,"i"],cmenu2,obj,faux]]);

menu.empile(["&Editer",["Couper",[FL_COMMANDE,"x"],cmenu4,obj,vrai],
                                ["&Copier",[FL_COMMANDE,"c"],cmenu3,obj,faux]]);
```

On rajoute ici deux éléments de menu : Fichier et Editer, chacun associé avec deux sous-menus.

Un sous-menu se décompose de la façon suivante:

a) Son nom: "&Nouveau Fichier"

b) Une combinaison de touches dont les valeurs sont les suivantes:

FL_MAJ	
FL_VERROUILLAGE_MAJ	
FL_CTRL	
FL_ALT	
FL_VERROUILLAGE_NUM	
FL_VERROUILLAGE_DEFILEMENT	
FL_COMMANDE	(uniquement sur Mac OS)
FL_CONTROLE	Equivalent à FL_CTRL

c) La fonction de rappel elle-même qui sera appelé

d) L'objet associé, qui sera passé avec la fonction de rappel

e) Vrai ou faux selon que l'on veuille rajouter un séparateur.

Une fois cette table encodé, on utilise la méthode *menu* de *fenêtre* pour l'enregistrer: *w.menu(menu,5,5,100,20);*

Déplacer un rectangle

Comme FLTK ne répond qu'à des événements clavier ou souris, une façon de dessiner des objets mouvants est d'introduire une tâche qui modifiera en arrière-plan les coordonnées de l'objet et induira la fenêtre à se redessiner.

```
classe mycoord {
	nombre x,y;

	fonction _initiale() {
		x=0;
		y=0;
	}
}

//nous déclarons notre objet
mycoord coords;

bool premier=vrai;
//Chaque fois la fenêtre sera modifiée, Cette fonction sera appelée avec l'objet o
fonction affichage(fenêtre w,mycoord o) {
	si (premier) {
		w.couleurdessin(FL_RED);
		w.dessinetexte("Press T",20,20);
	}
	sinon {
		w.stylecurseur(FL_CURSEUR_CROIX,FL_NOIR,FL_BLANC);
		w.couleurdessin(FL_RED);
		w.rectangle(o.x,o.y,60,60);
		//avec some text...
		w.dessinetexte("TEST",o.x+20,o.y+20);
	}
}

//Une fois lancée, cette tâche modifie les coordonnées du rectangle
//et force la fenêtre à se redessiner
tâche bouge(fenêtre wnd) {
	tantque (vrai) {
		coords.x++;
		coords.y++;
		wnd.redessine();
	}
}

//Si la touche « T » est pressée, on lance la tâche
fonction pressée(fenêtre w,chaine sclef,nombre iclef,mycoord o) {
	si (sclef=="T") {
		premier=faux;
		bouge(w);
	}
}



//Notre fenêtre avec objet et rappel
fenêtre wnd(coords) avec affichage;

//Instanciation de la fenêtre
```

```
wnd.commence(100,100,1300,900,"Dessin");
wnd.taillemax(10,10,0,0);
//Rappel sur touche pressée
wnd.sitouche(FL_TOUCHERELACHEE,pressée,coords);
wnd.fin();

wnd.lance();
```

Tâche: balles rebondissantes

```
nombre nb=0;
classe mycoord {

        nombre couleur;
        nombre x,y,ix,iy;
//commune signifie que ces valeurs sont communes à tous les objets
        commune nombre maxx,maxy;
        nombre idx;

        fonction _initiale(nombre xx,nombre yy) {
                couleur=FL_ROUGE;
                x=xx;
                y=yy;
                ix=1;
                iy=1;
                idx=nb;
                nb++;
        }

        fonction Idx() {
                renvoie(idx);
        }

        fonction increment() {
                x+=ix;
                si (x>=maxx)
                        ix=-1;
                sinon
                        si (x<=0)
                                ix=1;
                y+=iy;
                si (y>=maxy)
                        iy=-1;
                sinon
                        si (y<=0)
                                iy=1;
        }

        fonction RAZ() {
                x=10;
                y=10;
        }

        fonction X() {
                renvoie(x);
        }

        fonction Y() {
                renvoie(y);
        }
```

```
        fonction chaine() {
                chaine s=x+","+y;
                renvoie(s);
        }
}


//Ce tâche incrémente les coordonnées d'une balle
tâche déplace(fenêtre w,mycoord ballecoords) {
        tantque (vrai) {
                ballecoords.increment();
                tente {
                        w.redessine();
                }
                capte() {
                        renvoie;
                }
        }
}

//Nous créons un objet de base pour gérer la taille de la fenêtre
mycoord basecoords(0,0);
basecoords.maxx=500;
basecoords.maxy=300;
nombre debx,deby;

table balles;

fonction clicked(bouton b,fenêtre w) {
        //Les positions initiales sont aléatoires
        debx=aléatoire()*500;
        deby=aléatoire()*300;
        //nous créons notre nouvelle balle
        mycoord ballecoords(debx,deby);
        //nous enregistrons les coordonnées dans une table
        balles.empile(ballecoords);
        déplace(w,ballecoords);
}

fonction affichage(fenêtre w,table bs) {
        omni o;
        nombre i;
        //pour chaque balle, nous dessinons un simple cercle
        pour (o dans bs) {
                w.cercle(o.X(),o.Y(),10);
                w.dessinetexte(o.Idx(),o.X()-5,o.Y()+2);
        }
        //si les dimensions de la fenêtre ont changé, nous les utilisons comme
        //nouvelles contraintes
        basecoords.maxx=w.coords()[2];
        basecoords.maxy=w.coords()[3];
}


//Notre fenêtre
fenêtre wnd(balles) avec affichage;

wnd.commence(100,50,500,300,"Drawing");

wnd.taillemax(10,10,0,0);

//Création d'un bouton pour lancer une balle nouvelle
```

```
bouton b(wnd) avec clicked;
b.crée(10,10,20,20,FL_Régulier,FL_BOUTON_NORMAL,"Ok");


wnd.fin();
wnd.lance();
```

Créer des fenêtres dans une tâche

Il est possible de créer une fenêtre ou même de manipuler des objets graphiques depuis une tâche. En revanche, surtout lors de la création d'une fenêtre, il est important de protéger ces appels de façon à éviter les appels concurrents. Par exemple, il est préférable de construire une fenêtre dans une fonction *protégée* qui évitera que plusieurs tâches ne tentent de construire ces fenêtres en parallèle, brisant la continuité de la création d'une fenêtre.

Exemple

```
nombre px=300;
nombre py=400;
nombre nb=1;

// affichage d'un compteur
tâche bouge() {
        nombre i=0;
        fenêtre wx;
        sortie wo;
        chaine err;

//Définition d'un temps maximum au-delà duquel la tâche plante
        wx.tempsmax(0.1);
        //Verrouillage de la création
        verrouille("creation");
        tente {
                wx.commence(px,py,250,100,"ICI:"+nb);
                wo.crée(50,20,120,30,vrai,"Valeur");
                wx.fin();
                px+=300;
                nb++;
                si (px>=1800) {
                        px=300;
                        py+=150;
                }
        }
        capte(err) {
                //En cas d'erreur on ferme la fenêtre
                //uniquement si la fenêtre a été créée
                si (wx.créée())
                    wx.ferme();
                déverrouille("creation");
                renvoie;
        }
        //Désormais d'autres fenêtres peuvent être créées
        déverrouille ("creation");
        //Temps max différent pour l'affichage du compteur
        wo.tempsmax(1);
```

```
		tantque (vrai) {
			i++;
			tente {
				wo.valeur(i);
			}
			capte(err) {
				//Si temps dépassé, on ferme
				si ("Temps dépassé" dans err)
					continue;
				si (wx.créée())
					wx.ferme();
				renvoie;
			}
		}
}


fonction pressée(bouton b,omni n) {
		bouge();
}


fenêtre wnd;
wnd.commence(100,50,500,300,"Drawing");
bouton b avec pressée;
b.crée(430,20,60,60,FL_Regulier,FL_BOUTON_NORMAL,"Ok");
wnd.taillemax(10,10,0,0);
wnd.fin();

wnd.lance();
```

fparcourir (parcourir des chaines)

Cet objet permet d'afficher des chaines dans une boite et de les parcourir ou de les sélectionner individuellement ou en groupe.

Méthodes

1. ajoute(label): Ajoute une chaine au widget parcourir
2. aligne(nombre): définit l'alignement du label
3. cache(): Cache le widget
4. charge(filenom): Charge un fichier dans le widget parcourir
5. colle(): Colle chaine depuis le presse-papier
6. coords(nombre c): renvoie la table des coordonnées du widget
7. copie(chaine): Copie la chaine dans le presse-papier
8. couleurarrièreplan(nombre c|chaine c): initialise la couleur d'arrière-plan
9. couleurlabel(nombre c): définit ou renvoie la couleur du label
10. crée(x,y,largeur,hauteur,label): Crée un widget parcourir
11. créée(): Renvoie vrai si l'objet a été crée

12. désélectionne(): Désélectionne tous les noeuds
13. désélectionne(nombre i): Désélectionne le noeud i
14. efface(): Vide le widget parcourir de toutes ses valeurs
15. formatcarac(): Renvoie le caractère de formatage.
16. formatcarac(chaine): Définit le caractère de formatage
17. insère(l,label): Insère un label avant la ligne l
18. sélectionné(nombre i): Renvoie la chaine en position i
19. sélectionné(): Renvoie la chaine sélectionnée.
20. séparateurcolonne(): Renvoie le caractère de séparation de colonnes.
21. séparateurcolonne(chaine): Définit le caractère de séparation de colonnes
22. valeur(): renvoie la valeur courante sélectionnée

Sélection

La seule façon d'utiliser cet objet graphique, est de lui associer une fonction de rappel.

fonction parcourir_rappel(parcourir b,myobjet o);

Laquelle doit être déclarée avec l'opérateur « avec ».

Exemple

```
//la fonction de rappel
fonction rappel(parcourir b,omni n) {
	afficheligne("Selection:",b.select(),b.valeur());
}

fenêtre w;

w.commence(40,40,400,500,"Parcourir");

fparcourir b avec rappel;
b.crée(10,10,100,150,"Test");
b.ajoute("une");
b.ajoute ("deux");
b.ajoute ("trois");
b.ajoute ("quatre");

w.fin();
w.lance();
```

farbre et fnoeudarbre

Ces deux objets servent à manipuler un arbre dans lequel on peut sélectionner un ou des nœuds. Chaque nœud est « cliquable » et sa valeur est alors accessible via une fonction de rappel.

Méthodes de farbre

1. ajoute(chaine path): Ajoute un noeud.
2. ajoute(fnoeudarbre e,chaine n): Ajoute un noeud après e.
3. aligne(nombre): définit l'alignement du label
4. cache(): Cache le widget
5. cherche(chaine path): Renvoie l'élément correspondant au chemin path.
6. cliqué(): Renvoie l'élément qui a été cliqué.
7. colle(): Colle chaine depuis le presse-papier
8. coords(nombre c): renvoie la table des coordonnées du widget
9. copie(chaine): Copie la chaine dans le presse-papier
10. couleurarrièreplan(nombre c|chaine c): initialise la couleur d'arrière-plan
11. couleurconnecteur(nombre c): Définit ou renvoie la couleur du connecteur.
12. couleurlabel(nombre c): définit ou renvoie la couleur du label
13. couleurnoeudarrièreplan(nombre c): Définit ou renvoie la couleur d'arrière-plan du noeud.
14. couleurnoeudavantplan(nombre c): Définit ou renvoie la couleur d'avant-plan du noeud.
15. crée(nombre x,nombre y,nombre hauteur,nombre largeur,chaine label): Crée un arbre
16. créée(): Renvoie vrai si l'objet a été crée
17. Définit ou récupère l'espace (en pixels) qui doit apparaitre entre la bordure gauche du widget et le contenu de l'arbre.
18. dernier(): Renvoie le dernier élément.
19. efface(): Détruit les noeuds de l'arbre
20. estfermé(chaine path): Renvoie vrai si l'élément est fermé.
21. ferme(chaine path): Ferme l'élément.
22. ferme(fnoeudarbre e): Ferme l'élément.
23. focus(): Récupère le focus

24. insère(fnoeudarbre e,chaine label,nombre pos): Insère un élément après e avec label à la position pos dans la liste des enfants.
25. insèreavant(fnoeudarbre e,chaine label): Insère un élément au-dessus de e avec label.
26. estfermé(fnoeudarbre e): Renvoie vrai si l'élément est fermé.
27. label(chaine s): définit ou renvoie le texte du label
28. labelracine(chaine r): Définit le label de la racine.
29. largeurconnecteur(nombre s): Définit ou renvoie là largeur du connecteur.
30. margehaut(nombre s): Définit ou récupère l'espace (en pixels) qui doit apparaitre entre la bordure du haut du widget et le contenu de l'arbre.
31. modesélection(nombre s): Définit ou renvoie le mode de sélection.
32. montre(): Montre le widget
33. ordretri(nombre s): Définit ou renvoie l'ordre du tri.
34. ouvre(chaine path): Ouvre l'élément.
35. ouvre(fnoeudarbre e): Ouvre l'élément.
36. policelabel(nombre c): définit ou renvoie la police du label
37. policenoeud(nombre c): Définit ou renvoie la police du noeud.
38. précédent(fnoeudarbre e): Renvoie l'élément précédent avant e.
39. premier(): Renvoie le premier élément.
40. racine(): Renvoie la racine.
41. redessine(): Redessine la fenêtre
42. retire(fnoeudarbre e): Retire l'élément e dans l'arbre.
43. styleconnecteur(nombre s): Définit ou renvoie le style du connecteur.
44. suivant(fnoeudarbre e): Renvoie l'élément suivant e.
45. taillelabel(nombre c): définit ou renvoie la taille de la police du label
46. taillenoeud(nombre c): Définit ou renvoie la taille de la police du noeud.
47. tempsmax(décimal t): Définit le temps maximum pour cet objet
48. truc(chaine msg): Ajoute un message de 'truc' au widget
49. typelabel(nombre c): définit ou renvoie le type du label

Méthodes fnoeudarbre

1. active(): Active l'élément courant.
2. couleurarrièreplan(nombre c): Définit ou renvoie la couleur de l'arrière-plan du noeud.
3. couleuravantplan(nombre c): Définit ou renvoie la couleur de l'avant-plan du noeud.
4. désactive(): Désactive l'élément courant.
5. enfant(nombre i): Renvoie l'élément enfant à la position i.
6. enfants(): Renvoie le nombre d'enfants.
7. estactif(): Renvoie vrai si l'élément est actif.
8. estfermé(): Renvoie vrai si l'élément est fermé.
9. estouvert(): Renvoie vrai si l'élément est ouvert.
10. estracine(): Renvoie vrai si l'élément est une racine.
11. estsélectionné(): Renvoie vrai si l'élément est sélectionné.
12. label(): Renvoie le label du noeud.
13. raz(): Retire l'objet associé.
14. parent(): Renvoie le dernier élément.
15. police(nombre c): Définit ou renvoie la police du noeud.
16. précédent(): Renvoie l'élément précédent.
17. profondeur(): Renvoie la profondeur du noeud.
18. suivant(): Renvoie l'élément suivant.
19. taillepolice(nombre c): Définit ou renvoie la taille de police du noeud.
20. valeur(objet): Associe le noeud avec une valeur ou renvoie cette valeur.

Rappel

On peut évidemment associer un élément farbre avec une fonction de rappel dont la signature doit être la suivante :

fonction farbre_rappel(farbre t,fnoeudarbre i,nombre raison,monobjet o);

Cette fonction sera appelée chaque fois qu'un nœud sera sélectionné dans l'arbre. Raison peut avoir l'une des valeurs suivantes:

FL_ARBRE_RAISON_AUCUNE : pas de raison
FL_ARBRE_RAISON_SELECTIONNE : nœud sélectionné
FL_ARBRE_RAISON_DESELECTIONNE : nœud déselectionné
FL_ARBRE_RAISON_OUVERT : nœud ouvert
FL_ARBRE_RAISON_FERME : nœud fermé

Itérateur

Les objets farbre objet sont itératables.

Chemin

Certaines fonctions accèdent aux nœuds via un chemin dont la forme est similaire à celle d'un chemin UNIX :

Exemple: "/Racine/Sommet/sous-noeud"

Style connecteur

Le style des connecteurs entre noeuds est contrôlé par les valeurs suivantes:

FL_ARBRE_CONNECTEUR_AUCUN	Pas de lignes entre les nœuds.
FL_ARBRE_CONNECTEUR_POINTILLE	Lignes pointillées
FL_ARBRE_CONNECTEUR_SOLIDE	Lignes continues

Mode de sélection

La façon dont les nœuds sont sélectionnés dans l'arbre est contrôlée par les valeurs suivantes :

FL_ARBRE_SELECTION_AUCUNE	Rien n'est sélectionné quand un nœud est cliqué
FL_ARBRE_SELECTION_SIMPLE	Un seul nœud peut être sélectionné à la fois
FL_ARBRE_SELECTION_MULTI	Sélections multiples

Ordre de tri

Les nœuds peuvent être ajoutés à l'arbre selon les ordres de tri suivant :

FL_ARBRE_TRI_AUCUN	Pas de tri (défaut).
FL_ARBRE_TRI_ASCENDANT	Tri ascendant
FL_ARBRE_TRI_DESCENDANT	Tri descendant

Exemple

```
//Cette fonction est appelée quand un nœud est sélectionné ou déselectionné
fonction rappel(farbre t,fnoeudarbre i,nombre reason,omni n) {
	//nous changeons la taille de l'élément sélectionné
	si (reason==FL_ARBRE_RAISON_SELECTIONNE)
		i.taillepolice(20);
	sinon //l'élément désélectionné retrouve sa taille précédente.		si
(reason==FL_ARBRE_RAISON_DESELECTIONNE)
			i.taillepolice(FL_TAILLE_NORMAL);
}

fenêtre w;
farbre myarbre avec rappel;
fnoeudarbre ei;

w.commence(40,40,400,500,"Afficher");
mytree.crée(20,20,150,250,"Arbre");

mytree.labelracine("Racine");
ei=mytree.ajoute("Sous-racine);
mytree.ajoute(ei,"Test");
mytree.ajotue(ei,"Autre");
//nous ajoutons un nouvel élément comme chemin
mytree.ajoute("/Sous-racine/Nouveau noeud");
w.fin();

//nous modifions la police pour chaque élément après coup
//Ceci est équivalent à mytree.policenoeud(FL_TIMES_GRAS), avant d'ajouter
//les éléments
itérateur it=mytree;
pour (it.commence() tantque it.nfin() faire it.suivant())
	it.valeur().police(FL_TIMES_GRAS);


w.lance();
```

fentrée (zone de saisie)

Un objet fentrée définit une zone de saisie dans une fenêtre. Elle peut être utilisée avec une fonction de rappel qui sera appelé lorsque la zone de saisie sera détruite.

Méthodes

1. aligne(nombre): définit l'alignement du label
2. cache(): Cache le widget
3. colle(): Colle chaine depuis le presse-papier
4. coords(nombre c): renvoie la table des coordonnées du widget
5. copie(chaine): Copie la chaine dans le presse-papier

6. couleur(chaine c|nombre c): définit ou renvoie la couleur du texte
7. couleurarrièreplan(nombre c|chaine c): initialise la couleur d'arrière-plan
8. couleurlabel(nombre c): définit ou renvoie la couleur du label
9. couleursélection(nombre couleur): Couleur de l'élément sélectionné
10. crée(nombre x,nombre y,nombre largeur,nombre hauteur,booléen multiline,chaine label): Crée une fentrée
11. créée(): Renvoie vrai si l'objet a été crée
12. e[a:b]: Extrait des caractères entre a et b
13. e[a]: Extrait des caractères à la position a
14. focus(): Récupère le focus
15. insère(chaine s,nombre p): insère s à la position p dans la fentrée
16. label(chaine s): définit ou renvoie le texte du label
17. montre(): Montre le widget
18. mot(nombre pos): Renvoie le mot à la position pos
19. police(chaine s): définit ou renvoie la police du texte
20. policelabel(nombre c): définit ou renvoie la police du label
21. redessine(): Redessine la fenêtre
22. sélection(): renvoie le texte sélectionné dans la fentrée
23. taillelabel(nombre c): définit ou renvoie la taille de la police du label
24. taillepolice(nombre c): définit ou renvoie la taille de la police du texte
25. tempsmax(décimal t): Définit le temps maximum pour cet objet
26. truc(chaine msg): Ajoute un message de 'truc' au widget
27. typelabel(nombre c): définit ou renvoie le type du label
28. valeur()|(chaine v): renvoie le buffer d'fentrée ou définit le buffer initial

Exemple

```
classe block {
	//Nous déclarons notre fenêtre
	fenêtre w;
	chaine final;

	//A la fermeture de la fenêtre, result sera appelé
	//et nous pouvons alors récupérer le contenu de la zone
```

```
        //saisie…
        fonction result(fentrée txt,block bb) {
        //nous stockons le contenu de ce champ. dans une variable
                final=txt.valeur();
        }

        // Déclaration de notre champ fentrée
        //comme nous sommes au sein d'une classe, ici pointe sur l'instance
        //courante de cette classe
        fentrée txt(ici) avec result;

        fonction lancement() {
                w.commence(300,200,1300,150,"Modification");
                w.taillemax(10,20,0,0);
                //nous créons une fentrée multiligne
        txt.crée(200,20,1000,50,vrai,"Selection");
                //Initialisation
                txt.valeur("Un peu de texte");
                //Le texte sera en BLEU
                txt.couleur(FL_BLEU);
                // fin de session
                w.fin();
                //La taille de fentrée suivra celle de la fenêtre
                w.redimensionnable(txt);
                // lancement de la boucle principale
                w.lance();
        }
}

//Nous créons un block
block b;
//lancement
b.lancement();
//b.final contient la chaine qui a été tapée par l'utilisateur
afficheligne("Result:",b.final);
```

fsortie (Zone d'affiche de texte)

Ce type est utilisé pour afficher des données textuels non éditables dans une fenêtre.

Méthodes

1. aligne(nombre): définit l'alignement du label
2. couleur(chaine c|nombre c): définit ou renvoie la couleur du texte
3. couleurarrièreplan(nombre c|chaine c): initialise la couleur d'arrière-plan
4. couleurlabel(nombre c): définit ou renvoie la couleur du label
5. couleursélection(nombre couleur): Couleur de l'élément sélectionné

6. crée(nombre x,nombre y,nombre largeur,nombre hauteur,booléen multiline,chaine label): Crée une fsortie
7. créée(): Renvoie vrai si l'objet a été crée
8. police(chaine s): définit ou renvoie la police du texte
9. policelabel(nombre c): définit ou renvoie la police du label
10. taillepolice(nombre c): définit ou renvoie la taille de la police du texte
11. valeur(chaine s): définit le buffer de fsortie

boite (boite definition)

Ce type est utilisé pour dessiner une boite dans une fenêtre.

Méthodes

1. crée(nombre x,nombre y,nombre w,nombre h, chaine label): Crée une boite avec son label
2. type(nombre boxtype): modifie le type de la boite (voir ci-dessous pour une liste des types disponibles)

Type de boites

FL_PAS_DE_BOITE
FL_BOITE_PLATE
FL_BOITE_HAUTE
FL_BOITE_BASSE
FL_CADRE_HAUT
FL_CADRE_BAS
FL_BOITE_FINE_HAUTE
FL_BOITE_FINE_BASE
FL_CADRE_FIN_HAUT
FL_CADRE_FIN_BASE
FL_BOITE_GRAVEE
FL_BOITE_BOSSELEE
FL_CADRE_GRAVE
FL_CADRE_BOSSELE
FL_BOITE_BORDEE
FL_BOITE_OMBREE
FL_CADRE_BORDE
FL_CADRE_OMBRE
FL_BOITE_RONDE
FL_BOITE_ROMBREE
FL_CADRE_ROND
FL_BOITE_RPLATE
FL_BOITE_RONDE_HAUTE
FL_BOITE_RONDE_BASSE
FL_BOITE_DIAMANT_HAUTE
FL_BOITE_DIAMANT_BASSE
FL_BOITE_OVALE
FL_BOITE_OMBREEO

FL_CADRE_OVAL
FL_BOITE_PLATEO
FL_BOITE_PLASTIC_HAUTE
FL_BOITE_PLASTIC_BASSE
FL_CADRE_PLASTIC_HAUT
FL_CADRE_PLASTIC_BAS
FL_BOITE_PLASTIC_FINE_HAUTE
FL_BOITE_PLASTIC_FINE_BASSE
FL_BOITE_PLASTIC_RONDE_HAUTE
FL_BOITE_PLASTIC_RONDE_BASE
FL_BOITE_GTK_HAUTE
FL_BOITE_GTK_BASSE
FL_CADRE_GTK_HAUT
FL_CADRE_GTK_BAS
FL_BOITE_GTK_FINE_HAUTE
FL_BOITE_GTK_FINE_BASSE
FL_CADRE_GTK_FINE_HAUT
FL_CADRE_GTK_FINE_BAS
FL_BOITE_GTK_ROND_HAUTE
FL_BOITE_GTK_ROND_BASSE
FL_BOITE_TYPE_LIBRE

bouton

bouton est absolument fondamental dans la définition d'une interface graphique. Il fonctionne de pair avec une fonction de rappel dont la signature est la suivante :

```
fonction rappel_bouton(bouton b, myobj obj) {…}

bouton b(obj) avec rappel_bouton;
```

Il fournit les méthodes suivantes:

Méthodes

1. valeur(): renvoie la valeur du bouton courant
2. crée(nombre x,nombre y,nombre largeur,nombre hauteur,chaine type,chaine forme,chaine label): Crée un bouton. Si le type n'est pas fourni, il prend la valeur FL_Régulier par défaut.
3. image(wimage im,chaine label,nombre labelalign): Dessine le bouton avec une image particulière
4. label(chaine s): définit ou renvoie le texte du label
5. aligne(nombre): définit l'alignement du label du bouton
6. quand(chaine when1, chaine when2,...): Type de l'événement pour un bouton qui déclenche la fonction de rappel

7. raccourci(chaine clef code): Définit un raccourci-clavier correspondant à l'activation du bouton

Type de boutons

FL_Régulier
FL_Cocher
FL_Léger
FL_Répéter
FL_Renvoie
FL_Rond
FL_Image

Forme des boutons

FL_BOUTON_NORMAL
FL_BOUTON_INVERSEUR
FL_BOUTON_RADIO
FL_BOUTON_CACHE

Evénements (quand)

Voici la liste des événements qui peuvent être associés avec la fonction de rappel :

FL_QUAND_JAMAIS
FL_QUAND_CHANGE
FL_QUAND_RELACHE
FL_QUAND_RELACHE_TOUJOURS
FL_QUAND_TOUCHE_ENTRE
FL_QUAND_TOUCHE_ENTRE_TOUJOURS

Raccourcis

Voici la liste des raccourcis clavier qui peuvent être associés avec un bouton:

FL_Bouton
FL_Retour_Arrière
FL_Tab
FL_Entrée
FL_Pause
FL_Verrouillage_Défilement
FL_Echappe
FL_Accueil
FL_Gauche
FL_Haut
FL_Droit
FL_Bas
FL_Page_Haut
FL_Page_Bas

FL_Fin
FL_Imprimer
FL_Insère
FL_Menu
FL_Aide
FL_Verrouillage_Num
FL_KP
FL_KP_Entrée
FL_KP_Dernier
FL_F_Dernier
FL_Maj_G
FL_Maj_D
FL_Contrôle_G
FL_Contrôle_D
FL_Verrouillage_Maj
FL_Méta_G
FL_Méta_D
FL_Alt_G
FL_Alt_D
FL_Détruit

Exemple

```
classe block {
	fenêtre w;
	fentrée txt;
	chaine final;

	//Quand le bouton est pressé, Cette fonction est appelée
	//Elle ferme aussi la fenêtre courante
	fonction récupèretexte(bouton b,block bb) {
		final=txt.valeur();
		w.ferme();
	}


	fonction lancement(chaine ph) {
		final=ph;
		//Nous commençons l'instanciation de notre fenêtre
		w.commence(300,200,1300,150,"Modification");
		// notre fenêtre sera redimensionnable
		w.taillemax(10,20,0,0);

		txt.crée(200,20,1000,50,vrai,"Sélection");
		txt.valeur(ph);
		//Nous associons notre bouton avec la méthode récupèretexte
		bouton b(ici) avec récupèretexte;
		b.crée(1230,20,30,30,FL_Regulier,FL_BOUTON_NORMAL ,"Ok");
		// fin de session
		w.fin();
		w.redimensionnable(txt);
		// lancement
		w.lance();
	}
}
```

```
block b;
b.lancement("Ma phrase");
```

Image

Pour placer une image au centre d'un bouton, il suffit de suivre l'exemple ci-dessous.

```
image myimage;
//Nous chargeons une image GIF, avec un chemin de fichier
image.chargegif('c:\...');
//Nous déclarons notre bouton
bouton b avec récupèretexte;
//Pour lequel nous fournissons une image… LE TYPE EST :FL_Image
b.crée(1230,20,30,30,FL_Image,FL_BOUTON_NORMAL ,"Ok");
//Puis nous chargeons notre image dans notre bouton en précisant
//l'alignement…
b.image(myimage,"Dedans", FL_ALIGNE_CENTRE);
```

choix

kiffltk fournit un type particulier pour proposer des sélections dans une liste. Cet élément doit être initialisé avec un menu spécifique que nous décrirons plus loin.

Il fournit les méthodes suivantes :

Méthodes

1. valeur(nombre s): définit la valeur d'initialisation du choix
2. crée(nombre x,nombre y,nombre largeur,nombre hauteur,chaine label): Crée un choix
3. menu(table s): Initialise le menu
4. aligne(nombre): définit l'alignement du label

Menu

Une description de menu pour un choix est une table de tables.

```
vmenu=[["Premier",rappel1,"1"],["Second",rappel2,"2"],["Troisème",rappel3,"3"]];
```

Chaque élément est composé d'un nom, d'une fonction de rappel et d'un objet.

Menu Item: [nom,rappel,objet]

La fonction de rappel doit avoir la signature suivante:

fonction rappel_menu(choix c, myobjet obj);

Cette fonction est appelée pour chaque sélection de la liste.

Exemple

```
fenêtre w;

fonction rappel_menu(choix c, chaine s) {
        afficheligne(s);
}

table vmenu;
//Notre description de menu
vmenu=[["Premier",rappel_menu,"RRRR"],["second",rappel_menu,"OOOOOO"],["thir
d",rappel_menu,"BBBBBBB"]];

choix wch;

//nous créons notre fenêtre
w.commence(300,200,1300,500,"Fenetre");
//nous créons notre choix
wch.crée(20,420,100,50,"Choix");
wch.taillepolice(20);
//Cette opération doit être la dernière pour notre objet choix…
wch.menu(vmenu);
w.fin();
w.lance();
```

ftable

FLTK fournit un type particulier pour afficher et gérer des éléments dans une table : ftable.
Un objet de ce type doit être déclaré avec une fonction de rappel dont la signature est la suivante :

fonction rappel_table(ftable x,dictionnaire valeurs,myobjet obj);

ftable t(obj) avec rappel_table;

Les valeurs sont données via un dictionnaire, dont les clefs obéissent à la règle suivante selon la sélection effectuée dans la table:

“haut”:	la ligne du haut
“bas”:	la ligne du bas
“gauche”:	la colonne de gauche
“droit”:	la colonne de droite

“valeurs”: un dictionnaire, dont la clef est une chaine de type: “l:c”, avec l la ligne et c la colonne.

Ce objet fournit les méthodes suivantes :

Méthodes

1. ajoute(nombre R,nombre C,chaine v): Ajoute une valeur ligne R, colonne C
2. cellule(nombre R,nombre C): Renvoie la valeur de la cellule ligne R,colonne C
3. colonne()|(nombre nb): Définit le nombre de colonnes
4. crée(nombre x,nombre y,nombre largeur,nombre hauteur,chaine label): Crée une ftable d'objets, et commence l'ajout
5. entêtecolonne(nombre pos,chaine label): Définit le label de l'entête de la colonne à la colonne pos
6. entêteligne(nombre pos,chaine label): Définit le label de l'entête de la ligne à la ligne pos
7. hauteurentêteligne(nombre hauteur): la taille en pixel de l'entête de la ligne
8. hauteurligne(nombre hauteur): Définit la hauteur de la ligne en pixel
9. largeurcolonne(nombre largeur): Définit là largeur de la colonne en pixel
10. largeurentêtecolonne(nombre largeur): la taille en pixel de l'entête de la colonne
11. ligne()|(nombre nb): Définit le nombre de lignes
12. quand(chaine when1, chaine when2,...): Type de l'événement déclenchant la fonction de rappel
13. raz(): Nettoie la ftable
14. typeboite(nombre boitetype): type boite
15. typelabel(nombre c): définit ou renvoie le type du label

Exemple

```
fenêtre w;

fonction rappel_table(ftable x,dictionnaire V,fenêtre w) {
        afficheligne(V);
}

ftable t(w) avec rappel_table;
nombre i,j;
```

```
//nous créons notre fenêtre
w.commence(300,200,1300,500,"Fenetre");
//nous créons notre table
t.crée(20,20,500,400,"table");
t.taillepolice(12);
t.selectioncouleur(FL_BLEU);
//nous definissons la hauteur en pixels des lignes
t.hauteurligne(20);
//nous definissons la largeur en pixels des colonnes
t.largeurcolonne(60);
//nous remplissons notre table
pour (i=0;i<10;i++) {
        //y compris les entêtes
        t.ligneentête(i,"R"+i);
        t.colonneentête(i,"C"+i);
        pour (j=0;j<10;j++)
        //nous remplissons notre table avec des chaine de la forme: R0C9
                t.ajoute(i,j,"R"+i+"C"+j);
}
w.fin();
w.lance();
```

éditeur

KIFFLT fournit un type particulier pour gérer des données riches au sein d'un éditeur. Cet éditeur peut être associée avec une fonction de rappel dont la signature est la suivante.

```
fonction éditeurrappel(éditeur e,
        nombre pos,
        nombre ninsérés,nombre
        ndétruits,
        nombre mstyles,
        chaine déts,
        myobj obj);
```

Cette fonction est appelée chaque fois que le contenu de l'éditeur est modifié. Elle est associée à un éditeur lors de la déclaration de celui-ci :

```
éditeur e(obj) avec éditeurrappel;
```

Le arguments sont les suivants:

éditeur e: l'éditeur lui-même
pos: la position du curseur dans la fenêtre d'édition
ninsérés: le nombre de caractères insérés
ndétruits: le nombre de caractères détruits
mstyles: le nombre de caractères dont le style a été modifié

déts: les caractères qui viennent d'être détruit
obj: l'objet fourni lors de la déclaration.

Cet objet propose les méthodes suivantes:

Méthodes

1. accumule(chaine s): Ajoute s à la fin de l'entrée
2. ajoutestyle(dictionaire style): Initialise les styles
3. annote(chaine s|table v,chaine style,bool casse): Annote chaque occurence de s avec style
4. annoteregexip(chaine reg,chaine style): Annote chaque chaine correspondant à reg avec style
5. charge(chaine f): Charge le fichier dans l'éditeur
6. cherche(chaine s,nombre i,bool casse): Cherche la position de la sous-chaine s à partir de i
7. crée(nombre x,nombre y,nombre largeur,nombre hauteur,chaine label): Crée un éditeur
8. curseur(nombre l): renvoie la position du curseur courant du chariot ou définit la position du curseur
9. curseurcarac(): renvoie la position du curseur courant du chariot en nombre de caractères
10. désurbrille(): Retire la sur-brillance d'une portion de texte
11. détruit(): Détruit le texte sélectionné depuis l'éditeur
12. insère(chaine s,nombre p): insère s à la position p dans l'éditeur
13. lestyle(nombre start,nombre fin): renvoie une table de style pour chaque caractère de la section de texte
14. ligne(nombre l): renvoie la position de la ligne courante du chariot ou renvoie la ligne correspondant à la position
15. bornesligne(nombre l): renvoie le début et la fin de la position de la ligne courante du chariot ou renvoie les limites de la ligne correspondant à une position
16. borneslignecarac(): renvoie le début et la fin de la position de la ligne courante du chariot ou renvoie les limites de la ligne correspondant à une position en nombre de caractères
17. metstyle(nombre start,nombre fin,chaine style): définit le style d'une portion de texte
18. mot(nombre pos): Renvoie le mot à la position pos
19. positioncarac(nombre i): Convertit une position en caractère en une position en octet

20. positionoctet(nombre i): Convertit une position en octet en une position en caractère
21. rcherche(chaine s,nombre i,bool casse): Cherche la position de la sous-chaine s à partir de i en arrière
22. remplace(chaine s,chaine sub,nombre i,bool casse): remplace s avec sub selon la casse à partir de i
23. remplacetout(chaine s,chaine sub,bool casse): remplace toutes les occurrences de s avec sub selon la casse
24. sautautomatique(bool): Définit un mode coupure à la ligne automatique
25. sauvegarde(chaine f): Sauvegarde le contenu de l'éditeur dans le fichier f
26. sélection(): renvoie le texte sélectionné dans l'éditeur ou les coordonnées de la sélection selon la variable de réception
27. sidéfilementhorizontal(fonction f,objet o): fonction de rappel pour un défilement horizontal
28. sidéfilementvertical(fonction f,objet o): fonction de rappel pour un défilement vertical
29. sisouris(nombre action,fonction f,objet o): fonction de rappel après un clic
30. sitouche(nombre action,fonction f,objet o): fonction de rappel quand une touche est pressée
31. stylecurseur(nombre curseurforme): Définit la forme du curseur
32. surbrille(nombre start,nombre fin)|(): Met une portion de texte en sur-brillance. Renvoie vrai si le texte est sur-ligné ou le texte en sur-brillance
33. taillelabel(nombre c): définit ou renvoie la taille de la police du label
34. taillepolice(nombre c)|(): définit ou renvoie la taille de la police du texte
35. tailletexte(chaine l): Renvoie un dnombres avec largeur et hauteur comme clef pour donner les largeurs et hauteurs de la chaine en pixels
36. tempsmax(décimal t): Définit le temps maximum pour cet objet
37. truc(chaine msg): Ajoute un message de 'truc' au widget
38. valeur()|(chaine v): renvoie le texte dans l'éditeur ou initialise l'éditeur
39. valigne(nombre l,bool surbrille): Va à la ligne l. Si vrai, met la ligne en sur-brillance

Forme du curseur

KIFF offre plusieurs formes de curseur différentes :

FL_CURSEUR_NORMAL
FL_CARET_CURSEUR_CHARIOT
FL_CURSEUR_DIM
FL_CURSOR_BLOC
FL_CURSOR_LOURD

Utilisez stylecurseur pour choisir la forme qui vous intéresse.

Styles

Il est possible dans éditeur de mélanger polices, couleurs et tailles. Il faut cependant au préalable déclarer un dictionnaire de style de la forme suivante :

```
dictionnaire m={'#': [FL_NOIR,FL_COURIER,FL_TAILLE_NORMALE ],
        'A': [ FL_BLEU,FL_COURIER_GRAS,FL_TAILLE_NORMALE ]
        'B': [ FL_VERT_FONCE,FL_COURIER_ITALIQUE,FL_TAILLE_NORMALE ],
        'C': [ FL_VERT_FONCE, FL_COURIER_ITALIQUE,FL_TAILLE_NORMALE
],
        'D': [ FL_BLEU,FL_COURIER,FL_TAILLE_NORMALE ],
        'E': [ FL_ROUGE_FONCE,FL_COURIER,FL_TAILLE_NORMALE ],
        'F': [FL_ROUGE_FONCE,FL_COURIER_GRAS,FL_TAILLE_NORMALE ],
        };
```

Important: clef "#"

Le dictionnaire doit toujours avoir une clef "**#**", clef qui est utilisée pour définir le style par défaut. Si cette clef n'est pas fournie une erreur sera renvoyée.

Ce dictionnaire est transmis à l'éditeur via l'instruction: *ajoutestyle(m).*

On se sert des clefs pour indiquer à l'éditeur quel style appliquer sur une section de texte.

Exemple

```
//Le dictionnaire décrivant les styles disponibles au sein de notre éditeur
dictionnaire m={'#': [FL_NOIR,FL_COURIER,FL_TAILLE_NORMALE ],
        'A': [ FL_BLEU,FL_COURIER_GRAS,FL_TAILLE_NORMALE ]
        'B': [ FL_VERT_FONCE,FL_COURIER_ITALIQUE,FL_TAILLE_NORMALE ],
        'C': [ FL_VERT_FONCE, FL_COURIER_ITALIQUE,FL_TAILLE_NORMALE
],
        'D': [ FL_BLEU,FL_COURIER,FL_TAILLE_NORMALE ],
        'E': [ FL_ROUGE_FONCE,FL_COURIER,FL_TAILLE_NORMALE ],
```

```
        'F': [FL_ROUGE_FONCE,FL_COURIER_GRAS,FL_TAILLE_NORMALE ],
        };

fenêtre w;
éditeur e;

w.commence(300,200,1300,700,"Modification");
w.taillemax(10,20,0,0);

e.crée(200,220,1000,200,"Editor");
e.ajoutestyle(m);

e.valeur("Voilà un style intéressant");
//Nous appliquons le style 'C' sur la section de texte entre 14-26 : intéressant
e.metstyle(10,22,'C');
e.annote("l", 'E'); //chaque « l » reçoit le style E
w.fin();
w.lance();
```

Modification du style

Il est possible de redéfinir un style pour un éditeur donné. On doit pour cela rappeler une nouvelle fois la fonction ajoutestyle.

```
dictionnaire m={'#':[FL_NOIR,FL_COURIER,FL_TAILLE_NORMALE ],
        'truc':[ FL_ROUGE_FONCE,FL_COURIER,FL_TAILLE_NORMALE ]};

//Nous modifions « truc » pour lui donner une nouvelle description
fonction test(bouton b, éditeur e) {
    m["truc"]=[ FL_VERT_FONCE,FL_COURIER,FL_TAILLE_NORMALE ];
    e.ajoutestyle(m);
}

fenêtre w;
éditeur e;
bouton b(e) avec test;

w.commence(300,200,1300,700,"Modification");
w.taillemax(10,20,0,0);

e.crée(200,220,1000,200,"Editor");
e.ajoutestyle(m);
e.valeur("Voici un style intéressant");
e.metstyle(10,22,'truc');

b.crée(1230,20,30,30,FL_Régulier,FL_BOUTON_NORMAL,"Ok");

w.fin();
w.lance();
```

Trucs

Il est possible d'ajouter un « truc » à un style. Chaque fois que le curseur de la souris passera sur le texte ayant ce style, le message en

question s'affichera. Ainsi un item de style au lieu d'avoir trois éléments, en aura deux de plus :

- Message
- Couleur de fond du message

Pour obtenir une table ayant la forme suivante :

[couleur,police,taille,"Message",couleur de fond].

Si la couleur de fond n'est pas fournie, on utilise celle de l'item.

Exemple

```
dictionnaire m={'#':[FL_NOIR,FL_COURIER,FL_TAILLE_NORMALE ],
        'truc':[ FL_ROUGE_FONCE,FL_COURIER,FL_TAILLE_NORMALE,
        "Ceci est un TRUC",FL_JAUNE]};
```

Quand la souris passera sur les mots ayant le style *truc*, le message « Ceci est un truc » apparaitra en jaune à l'écran.

Rappels: défilement, souris et clavier

Les fonctions de rappel doivent avoir les signatures suivantes

Défilement

fonction vhscroll(éditeur e, objet n);

Souris

fonction souris_rappel(éditeur e, dnombres coords, objet n);

Le second paramètre est un dnombres avec les clefs suivantes:

coords["bouton"]	la valeur du dernier bouton pressé (1,2 ou 3)
coords["x"]	la coordonnée X dans la fenêtre de la souris
coords["y"]	la coordonnée Y dans la fenêtre de la souris
coords["xracine"]	la coordonnée absolue en X de la souris
coords["yracine"]	la coordonnée absolue en Y de la souris
coords["rouex"]	l'incrément sur l'axe X de la roue
coords["rouey"]	l'incrément sur l'axe Y de la roue

Clavier

fonction ctouche(éditeur e, chaine k, nombre iclef objet n);

Dans l'exemple qui suit, nous définissons trois fonctions de rappel, pour le défilement vertical, la souris et le clavier, de façon à afficher dans un objet fsortie la position de la ligne courante.

```
fonction cvscroll(éditeur e,fsortie num) {
        num.valeur(e.line());
}

fonction csouris(éditeur e,dnombres coords,fsortie num) {
        num.valeur(e.line());
}

fonction ctouche(éditeur e, chaine k, nombre i,fsortie num) {
        num.valeur(e.line());
}

fenêtre w;
éditeur e;
fsortie num;

w.commence(300,200,1300,700,"Fenêtre");
w.taillemax(10,20,0,0);
num.crée(100,100,30,40,”Line”);

e.crée(200,220,1000,200,"Editor");
e.sisouris(FL_RELEASE,csouris,num);
e.sidéfilementvertical(cvscroll,num);
e.sitouche(FL_CLEFUP,cclef,num);

w.fin();
w.lance();
```

défilement

Cet objet sert à définir une zone de défilement.

Méthodes

1. crée(nombre x,nombre y,nombre w,nombre h,chaine label): Crée une zone de défilement
2. redimensionnable(objet): rend l'objet redimensionnable

fprogression

Cette outil graphique permet d'afficher une barre de progression entre une valeur minimale et maximale.

On peut attache une fonction de rappel à un tel objet de façon à capturer les modifications de valeur.

Cette function doit avoir la signature suivante:

```
fonction rappel(fprogression s, omni obj) {
        //valeur() renvoie la valeur courante de la barre
        afficheligne(s.valeur());
}

fprogression s(obj) avec rappel;
```

Méthodes

1. **couleurarrièreplan (nombre c):** *définit ou renvoie la couleur d'arrière-plan*
2. **couleurbarre(chaine couleur|nombre couleur):** *Définit la couleur de la barre.*
3. **crée(nombre x,nombre y,nombre l,nombre h,nombre alignement, chaine label):** *Création de la barre*
4. **minimum(décimal x):** *définit ou renvoie le minimum*
5. **maximum(décimal x):** *définit ou renvoie le maximum*
6. **value(décimal):** *définit ou renvoie la valeur de la barre.*

Exemple:

```
fenêtre w;

fprogression c;

tâche progressons() {
    pour (nombre i dans <0,100,1>) {
        pour (nombre j dans <0,100000,1>) {}
        c.valeur(i);
    }
    afficheligne("Fin");
}


fonction Lancement(bouton b,omni e) {
    progressons();
}

bouton b avec Lancement;

w.commence(50,50,500,500,"test");
c.crée(30,30,300,30,"progression");
b.crée(100,100,50,50,"Ok");
c.minimum(0);
c.maximum(100);
```

```
c.couleurbarre(FL_BLEU);
w.fin();
w.lance();
```

fcompteur

kifltk offer deux sortes de compteur, l'un avec un niveau d'incrément l'autre avec deux.
Un fcompteur doit être attaché à une fonction de rappel de façon à capturer les modifications de valeur.

Cette function doit avoir la signature suivante:

```
fonction rappel(fcompteur s, omni obj) {
        //valeur() renvoie la valeur courante.
        afficheligne(s.valeur());
}

fcompteur s avec rappel;
```

Méthodes

1. **bornes(décimal x,décimal y):** *définit les limites du fcompteur*
2. **crée(nombre x,nombre y,nombre hauteur,nombre largeur,chaine label):** *Crée un fcompteur*
3. **police(nombre s):** *définit ou renvoie la police du texte*
4. **paslong(décimal):** *definit le pas long du fcompteur*
5. **pas(décimal):** *definit le pas du fcompteur*
6. **lespas(décimal court,décimal long):** *definit les pas court et long du fcompteur*
7. **couleurtexte(chaine code|nombre code):** *Définit la couleur du texte*
8. **tailletexte(chaine l):** *Renvoie un dictionnaire avec largeur et hauteur comme clef pour donner les largeurs et hauteurs de la chaine en pixels*
9. **type(bool normal):** *si 'vrai' alors fcompteur double sinon fcompteur simple*
10. **valeur(décimal):** *définit la valeur du fcompteur ou renvoie sa valeur*

Exemple:

```
fenêtre w;

fonction tst(fcompteur e,omni i) {
    afficheligne(e.valeur());
}

fcompteur c avec tst;

w.commence(50,50,500,500,"test");
c.crée(30,30,300,100,"Counter");
c.lespas(0.01,0.1);
c.tailletexte(20);
c.couleurtexte(FL_ROUGE);
w.fin();
w.lance();
```

glissière

KIFFLTK offre deux sortes de glissières (*sliders)*, dont l'un affiche en plus d'un curseur, la valeur correspondant à la position du curseur. La glissière doit être associée à une fonction de rappel dont la signature est la suivante:

```
fonction rappel_glissière(glissière s,myobj obj) {
        // valeur() renvoie la valeur courante du curseur
        afficheligne(s.valeur());
}

glissière s(obj) avec rappel_ glissière;
```

Voici les méthodes de cet objet :

Méthodes

7. aligne(nombre): définit l'alignement de la glissière
8. couleursélection (nombre couleur): Couleur pour les éléments sélectionnés
9. crée(nombre x,nombre y,nombre largeur,nombre hauteur,nombre alignement,bool valeurglissière,chaine label): Crée une glissière avec ou sans valeur
10. bornes(nombre x,nombre y): définit les limites de la glissière
11. pas(nombre): définit le pas de la glissière
12. type(nombre x): Type de la glissière
13. typeboite(nombre x): Définit le type de boite autour de la glissière

14. valeur(nombre): définit la valeur pour la glissière ou renvoie sa valeur

Type de glissière

FL_GLISSIERE_VERTICALE
FL_GLISSIERE_HORIZONTALE
FL_GLISSIERE_VERTICALE_PLEINE
FL_GLISSIERE_HORIZONTALE_PLEINE
FL_GLISSIERE_VERTICALE_JOLIE
FL_GLISSIERE_HORIZONTALE_JOLIE

Exemple

Cet exemple montre comment avec une glissière, on peut contrôler le mouvement d'un rectangle dans une fenêtre.

```
//Notre classe de coordonnées.
classe mycoord {

        nombre couleur;
        nombre x,y;

        fonction _initiale() {
                couleur=FL_ROUGE;
                x=0;
                y=0;
        }
}

mycoord coords;
fenêtre wnd(coords) avec affichage;

fonction glisser(glissière s,mycoord o) {
        //Nous récupérons sa valeur
        o.x=s.valeur();
        wnd.redessine();
}

glissière vs(coords) avec glisser;

wnd.commence(100,100,300,300,"Drawing");
wnd.taillemax(10,10,0,0);
//nous créons notre glissière
vs.crée(10,10,180,20,FL_ALIGNE_GAUCHE,vrai,"Position");
//les valeurs seront entre 0 et 300
vs.bornes(0,300);
//avec comme valeur initiale : 100
vs.valeur(100);

wnd.fin();
```

```
wnd.lance();
```

onglet et fgroupe

Ces objets permettent de gérer des fenêtres sous forme d'onglets. Chaque onglet correspond à une fenêtre dont la description est donnée sous la forme d'un fgroupe, correspondant peu ou prou à la façon dont on initialise une fenêtre.

Méthodes

onglet offre les méthodes suivantes:

1. ajoute(fgroupe g): Ajoute dynamiquement un nouveau onglet
2. aligne(nombre): définit l'alignement du label
3. cache(): Cache le widget
4. commence(nombre x,nombre y,nombre largeur, nombre hauteur,chaine titre): Crée un onglet fenêtre et commence l'initialisation
5. courant(): Renvoie l'onglet courant actif
6. fin(): fin de la création de l'onglet
7. retire(fgroupe g): Retire un onglet

Méthodes de fgroupe

L'objet fgroupe offre les méthodes suivantes:

1. active(): active l'onglet
2. commence(nombre x,nombre y,nombre w, nombre h,chaine tille): Crée un widget fgroupe et commence l'initialisation
3. fin(): fin de la construction du fgroupe.

Important

- La création d'un onglet est très simple. Il faut d'abord créer un onglet en fournissant la taille de sa boite extérieur. Puis dans votre onglet, vous créez autant de groupe que nécessaire.
- La dimension d'un groupe doit être inférieur à la hauteur de la boite de l'onglet.
- Chaque groupe doit avoir la même dimension.
- Les groupes suivants doivent toujours être cachés.

Rappel

Il est possible d'associer un fgroupe avec une fonction de rappel comme pour les objets fenêtres. Cette fonction de rappel est appelé chaque fois que le groupe doit être redessiné. La plupart des méthodes disponibles pour les fenêtres sont disponibles pour les groupes.

Simple exemple

Voici un exemple simple qui crée un bloc avec deux onglets.

```
fenêtre wnd;
//Nous créons notre fenêtre principale
wnd.commence(100,100,500,500,"Onglets");
//Une section onglet
onglet tabs;
//que nous définissons comme boîte.
tabs.commence(10,55,300,325,"Onglets");

//le premier groupe est une liste de widgets
fgroupe g1;
//nous commençons en chargeant notre widget
//La taille est 80=55+25 et la hauteur est 300=325-25
g1.commence(10,80,300,300,"Label&1");
//Nous y rajoutons une entrée
fentrée i1;
i1.crée(60,90,240,40,vrai,"Entrée 1");
//notre groupe 1 est maintenant construit
g1.fin();

//alors nous créons notre second onglet comme groupe de nouveau
fgroupe g2;
//la taille de ce groupe est la même que g1
g2.commence(10,80,300,300,"Label&2");
//IMPORTANT: nous le cachons
g2.cache();
//Nous ajoute notre nouveau widget
fentrée i2;
i2.crée(60,90,240,40,vrai,"Input 2");
//notre groupe 2 est maintenant achevé
g2.fin();
//ainsi que l'ensemble de nos onglets
tabs.fin();

wnd.fin();
wnd.lance();
```

Dans l'exemple qui suit, les onglets sont rajoutés ou retirés de façon dynamique.

```
nombre nb=0;

//Cette fonction détruira l'onglet courant
fonction retireonglet(bouton b, onglet t) {
//On récupère un pointeur sur l'onglet courant (de type fgroupe)
        omni x=t.courant();
        //On le retire
        t.retire(x);
}

//Cette fonction crée un nouvel onglet
fonction addtab(bouton b,wtabs x) {
```

```
        fgroupe g;
        //une seule taille pour tous
        g.commence(10,80,300,300,"Label&"+nb);
        //IMPORTANT: nous le cachons sauf si c'est le premier
        si (nb!=0)
                g.cache();
        //Nous ajoutons notre groupe
        fentrée i;
        i.crée(60,90,240,40,vrai,"Input "+nb);
        //Fin du groupe
        g.fin();
        nb++;
        //nous l'ajoutons à notre onglet...
        x.ajoute(g);
}


fenêtre wnd;
//Nous créons une fenêtre
wnd.commence(100,100,500,500,"TABS");
//puis un onglet
onglet tabs;
//dont nous définissons la boite
tabs.commence(10,55,300,325,"Onglets");
//Puis fin
tabs.fin();

//nous ajoutons un bouton pour déclencher la création d'un nouvel onglet
dynamiquement...
bouton b(tabs) avec addtab;
b.crée(400,100,50,30,FL_Régulier,FL_BOUTON_NORMAL,"Ajoute");

//Ce bouton détruit l'onglet courant
bouton br(tabs) avec retiretab;
br.crée(400,140,50,30,FL_Regulier, FL_BOUTON_NORMAL,"Retire");

wnd.fin();
wnd.lance();
```

Avec une fonction de rappel

```
//Notre fonction de dessin
fonction dessin(fgroupe w,omni n) {
    w.couleurdessin(FL_NOIR);
    w.cercle(100,100,100);
}

//nous créons un groupe lié à cette fonction de dessin
w_groupe fen avec dessin;
fen.commence(10,80,300,300,"Infos");
fen.fin();
//Nous l'ajoutons comme onglet
tabs.ajoute(fen);
```

fparcourirfichiers

Cet objet est utilisé pour choisir des fichiers depuis une fenêtre de sélection. Cet objet s'utilise avec une fonction de rappel dont la signature est la suivante :

fonction rappel_fichierparcourir(fparcourirfichier f,myobjet objet);

Les méthodes sont les suivantes :

Méthodes

1. crée(chaine intialdirectory,chaine filter,bool rép,chaine label): Ouvre un fparcourirfichiers, en mode répertoire si rép est vrai
2. ferme(): Ferme le widget fparcourirfichiers
3. ok(): renvoie vrai si ok a été sélectionné
4. ouvre() : Ouvre une fenêtre mac (uniquement dans l'interface graphique) et retourne le nom du fichier choisi
5. charge(chaine c) : Charge dans l'interface graphique mac un fichier.
6. valeur(): Renvoie le fichier sélectionné

Méthode

Il existe différentes méthodes pour ouvrir un fichier :

- FL_CHOIX_SIMPLE: ouvre fichier par fichier
- FL_CHOIX_MULTIPLE: ouverture multiple
- FL_CHOIX_SAUVEGARDE: sauvegarde
- FL_CHOIX_RÉPERTOIRE: pour choisir un répertoire

Exemple

```
//La fonction choisir permet de renvoyer la sélection
fonction choisir(fichierparcourir f,omni b) {
        si (f.ok()) {
                afficheligne("Ok:",f.valeur());
                b=vrai;
                f.ferme();
        }
}

bool b=faux;
fparcourirfichiers fb(b) avec choose;
```

```
fb.crée('C:\XIP',"*", FL_CHOIX_SIMPLE,"Choisir un fichier ");
```

Exemple de constructeur de bitmap

```
//Ce programme sert à produire des bitmaps à partir d'un dessin fait par l'utilisateur à la souris.

//Les dimensions de l'image... 16x16 (vous pouvez augmenter ces valeurs bien sûr)
nombre dx=64,dy=64;

classe coords {
    nombre  x,y;

    fonction _initiale() {
        x=20;
        y=20;
    }
}

dictionnaire pixels;
tablenombres lesbits;

//calcul du tableau de bits...
fonction calcul(bouton b,fenêtre f) {
    //Chaque élément de la table comprend 8 bits, soit des blocs de 8 pixels...
    nombre nb=dx/8;
    nombre sz=nb*dy;

    //Un peu d'haskell pour initialiser un tableau de sz 0
    lesbits=<0 | x <- [1..sz]>;

    nombre x,y,p;
    pour (omni o dans pixels.valeurs()) {
        x=o[2];
        y=o[3]*nb;
        p=x/8;
        x= 1<< (x%8);
        lesbits[y+p] = lesbits[y+p] | x;
    }
    //Une fois la table de bits calculée, on peut la redessiner
    f.redessine();
    afficheligne(lesbits);
}

//Ces paramètres permettent de contrôler la taille de la grille
nombre px=100,py=100, pw=dx*10,ph=dy*10;

//Cette fonction est associée avec la construction de la fenêtre...
fonction dessin(fenêtre f,coords o) {

    f.couleurdessin(FL_NOIR);
    pour (nombre i dans <px,px+pw+1,10>)
        f.ligne(i,px,i,px+pw);

    pour (nombre j dans <py,py+ph+1,10>)
        f.ligne(py,j,py+ph,j);
```

```
    //Nous avons la position de la souris
    //elle doit etre comprise entre px et px+pw
    si (o.x>=px et o.x<px+pw et o.y>=py et o.y<py+ph) {
        nombre xp,yp,i,j;
        xp=(o.x-px)/10;
        xp*=10;
        xp+=px;
        yp=(o.y-py)/10;
        yp*=10;
        yp+=py;

        //Un petit truc: On construit une clef sur la base de nos positions: elle est unique et
facilement retrouvable
        chaine k=xp+":"+yp;
        i=(xp-px)/10;
        j=(yp-py)/10;
        pixels[k]=[xp,yp,i,j];
    }

    pour (omni oo dans pixels.valeurs())
        f.rectangleplein(oo[0],oo[1],10,10,FL_ROUGE);

    //On affiche notre bitmap grandeur réelle
    si (lesbits.taille()) {
        bitmap btm;
        btm.charge(lesbits,dx,dy);
        f.bitmap(btm,FL_ROUGE,800,200);
    }
}

//Quand la souris bouge, cette fonction est automatiquement appelée
fonction gbougé(fenêtre f, dictionnaire cds, coords o) {
    //cds contient les informations propres au mouvement de la souris
    o.x=cds["x"];
    o.y=cds["y"];

    f.redessine();
}

//Variable principale pour garder la trace des coordonnées de notre rectangle
coords xy;

//On crée notre fenêtre
fenêtre f(xy) avec dessin;
bouton bt(f) avec calcul;

f.commence(120,120,1000,800,"Dessin");
//On offre deux actions selon que l'on maintient le bouton de la souris appuyé
f.sisouris(FL_RELACHE,gbougé,xy);
f.sisouris(FL_DRAGUE,gbougé,xy);

bt.crée(800,50,30,30,"Ok");
f.fin();
f.lance();
```

Type automate

Ce type fournit une méthode très efficace pour gérer des listes de mots. Une liste de mots se présente sous la forme d'un fichier dont chaque ligne correspond à un mot particulier.

Exemple:

Un fichier contenant une liste de mots aura le format suivant:

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

Un automate est très efficace pour déterminer si un mot appartient ou non à un lexique. Il peut enregistrer des centaines de milliers de mots tout en conservant de très bonnes performances. Il peut aussi être utilisé conjointement avec une fonction pour contrôler la façon dont cet automate est parcouru.

Méthodes

Il offre les méthodes suivantes:

1. **ajoute(chaine mot):** *ajoute un mot à l'automate*
2. **ajoute(chaine mot,chaine lemme):** *ajoute un mot et sa forme lemmatisée dans l'automate. Le lemme peut aussi être associé avec des traits linguistiques.*
3. **dump(chaine chemin):** *dump le contenu de l'automate en mémoire dans un fichier. Particulièrement utile lorsque de nombreux mots ont été enregistrés via* ajoute.
4. **distanceédition(chaine mot,nombre seuil,nombre critères):** *Calcule une distance d'édition sur la base de l'automate selon*

un seuil donné.. Les critères *sont une combinaison des actions suivantes:*

a_premier: *Le premier caractère du mot peut être modifié selon les autres actions. Par défaut, on évite de modifier le premier caractère d'un mot pour limiter le temps de recherche.*

a_change: *L'automate peut modifier un caractère en un autre.*

a_détruit: *L'automate peut détruire un caractère*

a_insère: *L'automate peut insérer un caractère*

a_commute: *L'automate commute deux caractères consécutifs.*

a_sanscasse: *L'automate peut prendre en compte la casse (majuscule/minuscule) lors de la modification d'un caractère en un autre et dans ce cas attribuer un poids de 0.1 au lieu de 1.*

a_surface: *L'automate renvoie la forme de surface dans un transducteur.*

a_tout: *L'automate renvoie la forme de surface dans un transducteur concaténée au lemme. Le caractère* caractèretrait *lorsqu'il est fourni est utilisé comme séparateur.*

a_découpe : *Permet le découpage d'une chaine comprenant plusieurs mots en une liste de ces mots.*

a_saute : *Saute les caractères d'une chaine inconnue dans l'automate. A utiliser en conjonction avec a_découpe pour éliminer les caractères inconnus. Attention à la différence de a_détruit, a_saute n'alourdit pas le score.*

a_trace: *Rajoute un autre dictionnaire, qui conserve pour chaque mot la liste des modifications nécessaires pour le produire.*

5. **charge (chaine chemin,chaine caractèretrait):** *Charge un fichier dont chaque ligne comprend un mot.* caractèretrait *qui est optionnel définit le caractère qui introduit des traits.*

6. **chargelemmes(chaine chemin,chaine caractèretrait):** *Charge un fichier dont les lignes fonctionnent par pair. La ligne paire contient la forme de surface et la ligne impaire la forme lemmatisée.* caractèretrait *qui est optionnel définit le caractère qui introduit des traits ou sert de séparateur avec l'option* a_tout.

7. **chargecompact(chaine chemin,chaine caractèretrait):** *Charge un fichier en format compact (voir* enregistre *plus bas)* caractèretrait *qui est optionnel définit le caractère qui introduit des traits ou sert de séparateur avec l'option* a_tout.

8. **recherche(chaine mot):** *Recherche un mot dans l'automate en conjonction avec la fonction de modèle.*

9. **recherche(chaine mot,omni o):** *Recherche un mot dans l'automate en conjonction avec la fonction de modèle. Cette méthode fournit aussi un objet utilisateur qui apparaitre en fin de liste de paramètre de la fonction.*

10. **modèle(fonction):** *Définit la fonction modèle. Cette fonction peut être aussi déclarée avec l'opérateur « avec » à la déclaration de l'automate.*

11. **définiscodetrait(chaine c):** *Définit le caractère qui sert de frontier aux traits. Retrourne une erreur lorsqu'il a déjà été défini.*

12. **initialisetransformations(dictionnaire m):** *Initialise le dictionnaire des poids à utiliser lors d'un distanceédition dans l'automate*

13. **enregistre(chaine chemin):** *Enregistre un fichier en mode compact qui permet une lecture et un stockage plus efficace.*

Déclaration

Un automate peut être déclaré avec une fonction dite « fonction modèle" et un objet. Cette fonction est utilisée pour contrôler le fonctionnement de l'automate caractère par caractère.

Deux types de fonction sont disponibles avec les signatures suivantes:

a) La plus simple :

fonction simple(nombre c,tabnombres labels,décimal score)

où:

a) c est le caractère en format *Unicode* courant (s'il est négatif, c'est que c'est le premier de la chaine)

b) labels est la liste des caractères potentiels suivants.

c) score est le score courant.

b) La version plus complexe est la suivante :

```
fonction cpl(chaine machaine,
        chaine base,
        nombre courant,
        tablechaines labels,
        tablechaines actions,
        décimal score,
        omni o)
```

où:

a) machaine est la chaine analysée

b) base est la chaine extraite jusque là

c) courant est la position du curseur dans l'analyse de machaine.

d) labels est une table des caractères potentiels suivants selon l'automate.

e) actions est une table des actions effectuées jusque là. Il ne contient pas en revanche les caractères non modifés.

f) score est le score courant

g) o est un objet utilisateur fourni soit à la déclaration soit via *recherche.*

Valeur de retour: dicocd ou table de tables

a) Ces deux fonctions peuvent renvoyer un dictionnaire de décimaux, où chaque clef correspond aux caractères retenus parmi les labels fournis à la fonction. Dans le cas de la fonction simple, il faut transformer les les clefs en chaine avec la méthode *ord, avant d'être stocké dans l'objet dicocd.* Lorsque vous construisez ce *dicocd*, vous devez impérativement utiliser les clefs provenant de *labels.* Vous pouvez aussi supprimer le caractère courant en utilisant comme clef « ~ ». Si en revanche vous voulez forcer l'automate à rajouter une valeur, il faut concaténer « ~ » avec le caractère en question.

dicocd fm={‘a’:0.9,’b’:0.8}

b) Elles peuvent aussi renvoyer une table de tables, ou chaque sous-table contient l’action à effectuer et son score.
table t=[[‘a’,0.9],[‘b’:0.8]];

La différence fondamentale entre ces deux approches est le fait que dans le deuxième cas, les actions sont ordonnées, ce qui n’est pas le cas dans le premier cas. Il devient possible alors de couper dans l’analyse selon des critères différents.

Processus

Le processus est plutôt simple. La fonction « recherche » traverse la chaine caractère par caractère. Par exemple, la chaine “zone” sera analysée de la façon suivante. Le premier caractère est “z”. Le système cherche alors dans l’automate un état initial correspondant à cette lettre. Si un tel état existe, il appellera notre fonction modèle, plaçant dans *labels* les arcs suivants, autrement dit les lettres pouvant un suivre un « z » dans un dictionnaire. La fonction déterminera laquelle de ces lettres sera conservée qu’elle renverra dans un *dicocd.*

z – i – n – g
o – n – e
– a – l

Nous avons enregistré dans notre automate les trois mots suivants: *zing, zone* and *zonal*.

Nous cherchons à corriger le mot suivant: *zong*.

Nous commençons avec “z”, la première lettre de notre mot. Dans ce cas dans notre automate seul l’arc « z » existe, par conséquent l’automate renvoie: labels=[‘z’].

Caractère suivant: ‘o’.

Après un « z » dans notre automate, seul un « i » et un « o » sont possibles.

Le système rappelle alors notre fonction avec: labels=[‘o’,’i’].

Nous analysons ces labels et quatre options s’ouvrent à nous:

a) Le « o » est correct. C'est à la fois un caractère de notre chaine et il est aussi dans l'automate. Nous l'ajoutons alors à notre résultat. Le score n'est pas modifié.

b) Le « o » est incorrect. Cela pourrait être un « i ».Cependant comme les deux caractères sont différents (nous avons un « o » dans la chaine) nous infligeons à ce choix une pénalité de 1 : res["i"]=1.

c) Le « o » est faux et doit être détruit. Nous ajoutons: res["~"]=score+1, puisqu'il y a là aussi une pénalité par rapport à notre chaine.

d) Nous pourrions ajouter un « i » ou « o » supplémentaire à notre chaine, alourdissant encore le score : res["~i"]=score+1 and res["~o"]=score+1.

Ce qui nous donne comme résultat à renvoyer: res={"o":0, "i":1, "~":1, "~i":1, "~o":1}.

La fonction *recherche* va alors utiliser ces informations pour continuer à explorer l'automate.

i. Si elle choisit "i" comme caractère suivant, alors le score sera de 1, et le chemin suivant "ng" correspondra à un score de 0.

ii. Si elle choisit "o", alors la pénalité sera de 1 et "e" and "g" seront différents ("zon*e*" and "zon*g*").

En choisissant d'autres chemins dans l'automate, on pourrait encore alourdir le score.

Déclaration

On déclare ces fonctions soit via une déclaration de l'automate comme ci-dessous, soit via *modèle.*

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

Opérateur *dans*

L'opérateur permet de détecter si un mot appartient ou non à l'automate, indépendemment de la fonction modèle.

Traits linguistiques

Des traits peuvent être associés au mot. Ils ne sont pas intégrés dans la détection mais sont renvoyés lors du résultat final.

Exemple

automate au;

```
au.définiscodetrait("+");
au.ajoute("zonking+Sg+Noun");// +Sg+Noun sera isolé de « zonking »
```

Distance d'édition

La fonction d'édition peut être associée avec certaines actions, dont la combinaison peut sérieusement ralentir la recherche.

Exemple

```
automate au;

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

Exemple

```
//La plus simple qui soit
fonction simple(nombre c,tabnombres labels,décimal score) {
    dicocd res;
    //Score trop grand, on repart.
    si (score>2)
        renvoie(res);

    nombre i;
    chaine a;
    bool début=faux;
    // Une valeur négative pour c signifie que c'est le premier caractère
    si (c<0) {
        début =vrai;
        c*=-1;
    }

    //Si le premier caractère est dans label, on le garde dans notre dicocd.
    si (début et c dans labels)
        res[c.car()]=0;
    sinon {
        //Nous voulons un score pour chaque conservé
        //Nous sautons le caractère suivant
        res["~"]=score+1;
        pour (i dans labels) {
```

```
            //Ici nous devons transformer notre code en chaine
            a=i.car();
            //Nous composant un rajout de caractère
            res["~"+a]=score+1;
            //Si le caractère suivant est différent, nous le remplaçons
            si (i!=c)
                //Remplacement
                res[a]=score+1;
            sinon
                //Même caractère, on ne fait rien
                res[a]=score;
        }
    }
    //Nous renvoyons à l'automate ce dictionnaire pour qu'il continue son
   //parcours dans l'automate.
    renvoie(res);
}

//Le mot à vérifier
chaine ch="zannking";

//Nous déclarons cet automate avec notre fonction
automaton au(ch) with simple;
//Nous chargeons notre fichier de mots.
au.charge('englishlexicon.txt');
//Nous vérifions s'il appartient à l'automate
si ("zonking" dans au)
    afficheligne("Ok");

//La fonction retourne un dicocd, contenant les mots les plus proches
affichelIgne(au.recherche(ch));
```

son

KIFF fournit aussi un jeu d'instruction pour jouer de nombreux types de fichiers audios (WAV, MP3, FLAC, OGG etc.). Vous chargez simplement votre fichier et vous pouvez le jouer n'importe où dans votre code. Le type fourni s'appelle simplement : « son ».

N.B. kiff se base sur *libao4, libsndfile-1* et *libmpg123* pour le décodage et l'exécution des fichiers audios.

Méthodes

Les méthodes proposées sont les suivantes:

1. **charge(chaine fichier):** *charge le fichier son*
2. **décode(tablenombres bufferson):** *decode le fichier « audio » buffer par buffer. Renvoie « faux » quand la fin du fichier a été atteinte.*
3. **encode(tablenombres bufferson):** *joue un buffer "audio" renvoyé par* décode.
4. **ferme():** *ferme le flux audio*
5. **interromps():** *arrête le flux audio. Il est nécessaire d'avoir lancé le fichier audio dans une tâche, pour pouvoir utiliser cette instruction.*
6. **joue():** *joue le flux audio*
7. **joue(bool début) :** *rejoue le fichier depuis le début.*
8. **joue(tablenombres bufferson) :** *joue un buffer « audio ».*
9. **ouvre(dictionnaire params):** *Ouvre un flux audio avec les paramètres du fichier son tel que renvoyé par* paramètres.
10. **paramètres():** *renvoie les paramètres du fichier audio sous la forme d'un dictionnaire.*
11. **paramètres(dictionnaire modifs):** *Seuls "taux" et "canaux" peuvent être modifés.*
12. **raz():** *réinitialise le fichier "audio" au début.*

Vous pouvez aussi charger le son à la declaration de l'objet en question.

Exemple

```
son s;

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

```
s.joue();
```

La declaration suivante est équivalente aux deux premières lignes:

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

Exemple de décodage

```
//On ouvre un fichier audio
son s('C:\XIP\XIP7\sound\Kalimba.mp3');

//On ouvre un second flux audio
son c;

//On récupère les paramètres du fichier audio
dictionnaire params=s.paramètres();

//que l'on utilise pour ouvrir un nouveau flux audio
c.ouvre(params);

//Nous bouclons dans notre fichier audio
//et pour chaque nouveau buffer, nous le jouons
tablenombres snd;
tantque (s.décode(snd))
    c.encode(snd);

//Nous fermons notre flux
c.ferme();
```

curl (Chargement de pages WEB)

Le type curl est utilisé pour charger des pages HTML depuis internet. Il est basé sur la bibliothèque (http://curl.haxx.se/) et offre quelques outils pour gérer des pages HTML.

Chargement de kiffcurl

Note : *kiffcurl* ne fait pas partie des bilbiothèques intégrées par défaut dans la machine virtuel *KIFF.* Il faut la charger explicitement via l'opérateur *importe* : *importe("kiffcurl");*

En cas d'erreur

Il se peut que la machine proteste et affiche le message suivant : *KIFF(999): Veuillez définir KIFFLIBS (ligne:1)*

- Dans ce cas, soit vous fournissez le chemin complet de votre bibliothèque : *importe('/monchemin/malib/kiffcurl') ;*

- Soit si vous désirez charger plus d'une bibliothèque dont le chemin vous est connu, vous pouvez alors initialiser le chemin via : *kifsys.env('KIFFLIBS', "votre chemin") ;*
  - Par la suite, le *importe('kiffcurl') ;* devrait se charger sans difficultés.

Méthodes

1. exécute() : pour exécuter une chaine de commande curl. Les options doivent avoir été fournies au préalable via *options.*
2. exécute(chaine nomfichier) : pour exécuter une chaine de commande curl. Les options doivent avoir été fournies au préalable via *options.* Le résultat de l'exécution est placé dans un fichier.
3. options(chaine option, chaine|nombre paramètre) : *pour fournir les options dont curl a besoin pour certains traitements. Voir plus bas, pour une liste complète de ces options.*
4. proxy(chaine proxy): connexion via un proxy. Cet appel n'est pas obligatoire hors du monde de l'entreprise.

5. session(chaine utilisateur,chaine psswrd): pour fournir au site un mot de passe et un nom d'utilisateur. Cet appel n'est pas obligatoire hors du monde de l'entreprise.
6. url(chaine html): Pour charger une page via une adresse URL. Cette commande effectue en fait un *options("CURLOPT_URL",uri)* avant de lancer l'exécution elle-même.
7. url(chaine html,chaine fichiernom): pour enregistrer une page dans un fichier.

Options

CURLOPT_ACCEPTTIMEOUT_MS,CURLOPT_ACCEPT_ENCODING,CURLOPT_ADDRESS_SCOPE,
CURLOPT_APPEND,CURLOPT_AUTOREFERER,CURLOPT_BUFFERSIZE,
CURLOPT_CAINFO,CURLOPT_CAPATH,CURLOPT_CERTINFO,
CURLOPT_CHUNK_BGN_FUNCTION,CURLOPT_CHUNK_DATA,CURLOPT_CHUNK_END_FUNCTION,
CURLOPT_CLOSESOCKETDATA,CURLOPT_CLOSESOCKETFUNCTION,CURLOPT_CONNECTTIMEOUT,
CURLOPT_CONNECTTIMEOUT_MS,CURLOPT_CONNECT_ONLY,CURLOPT_CONV_FROM_NETWORK_FUNCTION,
CURLOPT_CONV_FROM_UTF8_FUNCTION,CURLOPT_CONV_TO_NETWORK_FUNCTION,CURLOPT_COOKIE,
CURLOPT_COOKIEFILE,CURLOPT_COOKIEJAR,CURLOPT_COOKIELIST,
CURLOPT_COOKIESESSION,CURLOPT_COPYPOSTFIELDS,CURLOPT_CRLF,
CURLOPT_CRLFILE,CURLOPT_CUSTOMREQUEST,CURLOPT_DEBUGDATA,
CURLOPT_DEBUGFUNCTION,CURLOPT_DIRLISTONLY,CURLOPT_DNS_CACHE_TIMEOUT,
CURLOPT_DNS_SERVERS,CURLOPT_DNS_USE_GLOBAL_CACHE,CURLOPT_EGDSOCKET,
CURLOPT_ERRORBUFFER,CURLOPT_FAILONERROR,CURLOPT_FILETIME,
CURLOPT_FNMATCH_DATA,CURLOPT_FNMATCH_FUNCTION,CURLOPT_FOLLOWLOCATION,
CURLOPT_FORBID_REUSE,CURLOPT_FRESH_CONNECT,CURLOPT_FTPPORT,
CURLOPT_FTPSSLAUTH,CURLOPT_FTP_ACCOUNT,CURLOPT_FTP_ALTERNATIVE_TO_USER,
CURLOPT_FTP_CREATE_MISSING_DIRS,CURLOPT_FTP_FILEMETHOD,CURLOPT_FTP_RESPONSE_TIMEOUT,
CURLOPT_FTP_SKIP_PASV_IP,CURLOPT_FTP_SSL_CCC,CURLOPT_FTP_USE_EPRT,
CURLOPT_FTP_USE_EPSV,CURLOPT_FTP_USE_PRET,CURLOPT_GSSAPI_DELEGATION,
CURLOPT_HEADER,CURLOPT_HEADERDATA,CURLOPT_HEADERFUNCTION,
CURLOPT_HTTP200ALIASES,CURLOPT_HTTPAUTH,CURLOPT_HTTPGET,
CURLOPT_HTTPHEADER,CURLOPT_HTTPPOST,CURLOPT_HTTPPROXYTUNNEL,
CURLOPT_HTTP_CONTENT_DECODING,CURLOPT_HTTP_TRANSFER_DECODING,CURLOPT_HTTP_VERSION,
CURLOPT_IGNORE_CONTENT_LENGTH,CURLOPT_INFILESIZE,CURLOPT_INFILESIZE_LARGE,
CURLOPT_INTERLEAVEDATA,CURLOPT_INTERLEAVEFUNCTION,CURLOPT_IOCTLDATA,
CURLOPT_IOCTLFUNCTION,CURLOPT_IPRESOLVE,CURLOPT_ISSUERCERT,
CURLOPT_KEYPASSWD,CURLOPT_KRBLEVEL,CURLOPT_LOCALPORT,
CURLOPT_LOCALPORTRANGE,CURLOPT_LOW_SPEED_LIMIT,CURLOPT_LOW_SPEED_TIME,
CURLOPT_MAIL_FROM,CURLOPT_MAIL_RCPT,CURLOPT_MAXCONNECTS,
CURLOPT_MAXFILESIZE,CURLOPT_MAXFILESIZE_LARGE,CURLOPT_MAXREDIRS,
CURLOPT_MAX_RECV_SPEED_LARGE,CURLOPT_MAX_SEND_SPEED_LARGE,CURLOPT_NETRC,
CURLOPT_NETRC_FILE,CURLOPT_NEW_DIRECTORY_PERMS,CURLOPT_NEW_FILE_PERMS,
CURLOPT_NOBODY,CURLOPT_NOPROGRESS,CURLOPT_NOPROXY,
CURLOPT_NOSIGNAL,CURLOPT_OPENSOCKETDATA,CURLOPT_OPENSOCKETFUNCTION,
CURLOPT_PASSWORD,CURLOPT_PORT,CURLOPT_POST,
CURLOPT_POSTFIELDS,CURLOPT_POSTFIELDSIZE,CURLOPT_POSTFIELDSIZE_LARGE,
CURLOPT_POSTQUOTE,CURLOPT_POSTREDIR,CURLOPT_PREQUOTE,
CURLOPT_PRIVATE,CURLOPT_PROGRESSDATA,CURLOPT_PROGRESSFUNCTION,
CURLOPT_PROTOCOLS,CURLOPT_PROXY,CURLOPT_PROXYAUTH,
CURLOPT_PROXYPASSWORD,CURLOPT_PROXYPORT,CURLOPT_PROXYTYPE,
CURLOPT_PROXYUSERNAME,CURLOPT_PROXYUSERPWD,CURLOPT_PROXY_TRANSFER_MODE,
CURLOPT_PUT,CURLOPT_QUOTE,CURLOPT_RANDOM_FILE,
CURLOPT_RANGE,CURLOPT_READDATA,CURLOPT_READFUNCTION,
CURLOPT_REDIR_PROTOCOLS,CURLOPT_REFERER,CURLOPT_RESOLVE,
CURLOPT_RESUME_FROM,CURLOPT_RESUME_FROM_LARGE,CURLOPT_RTSP_CLIENT_CSEQ,
CURLOPT_RTSP_REQUEST,CURLOPT_RTSP_SERVER_CSEQ,CURLOPT_RTSP_SESSION_ID,
CURLOPT_RTSP_STREAM_URI,CURLOPT_RTSP_TRANSPORT,CURLOPT_SEEKDATA,
CURLOPT_SEEKFUNCTION,CURLOPT_SHARE,CURLOPT_SOCKOPTDATA,
CURLOPT_SOCKOPTFUNCTION,CURLOPT_SOCKS5_GSSAPI_NEC,CURLOPT_SOCKS5_GSSAPI_SERVICE,
CURLOPT_SSH_AUTH_TYPES,CURLOPT_SSH_HOST_PUBLIC_KEY_MD5,CURLOPT_SSH_KEYDATA,
CURLOPT_SSH_KEYFUNCTION,CURLOPT_SSH_KNOWNHOSTS,CURLOPT_SSH_PRIVATE_KEYFILE,
CURLOPT_SSH_PUBLIC_KEYFILE,CURLOPT_SSLCERT,CURLOPT_SSLCERTTYPE,
CURLOPT_SSLENGINE,CURLOPT_SSLENGINE_DEFAULT,CURLOPT_SSLKEY,
CURLOPT_SSLKEYTYPE,CURLOPT_SSLVERSION,CURLOPT_SSL_CIPHER_LIST,
CURLOPT_SSL_CTX_DATA,CURLOPT_SSL_CTX_FUNCTION,CURLOPT_SSL_SESSIONID_CACHE,
CURLOPT_SSL_VERIFYHOST,CURLOPT_SSL_VERIFYPEER,CURLOPT_STDERR,
CURLOPT_TELNETOPTIONS,CURLOPT_TFTP_BLKSIZE,CURLOPT_TIMECONDITION,
CURLOPT_TIMEOUT,CURLOPT_TIMEOUT_MS,CURLOPT_TIMEVALUE,
CURLOPT_TLSAUTH_PASSWORD,CURLOPT_TLSAUTH_TYPE,CURLOPT_TLSAUTH_USERNAME,
CURLOPT_TRANSFERTEXT,CURLOPT_TRANSFER_ENCODING,CURLOPT_UNRESTRICTED_AUTH,
CURLOPT_UPLOAD,CURLOPT_URL,CURLOPT_USERAGENT,
CURLOPT_USERNAME,CURLOPT_USERPWD,CURLOPT_USE_SSL,

**CURLOPT_VERBOSE,CURLOPT_WILDCARDMATCH,CURLOPT_WRITEDATA,
CURLOPT_WRITEFUNCTION,CURLOPT_UNIX_SOCKET_PATH,CURLOPT_XFERINFODATA,
CURLOPT_XFERINFOFUNCTION,CURLOPT_XOAUTH2_BEARER,CURLOPT_SSL_ENABLE_ALPN,
CURLOPT_SSL_ENABLE_NPN,CURLOPT_SSL_FALSESTART,CURLOPT_SSL_OPTIONS,
CURLOPT_SASL_IR,CURLOPT_SERVICE_NAME,CURLOPT_PROXYHEADER,
CURLOPT_PATH_AS_IS,CURLOPT_PINNEDPUBLICKEY,CURLOPT_PIPEWAIT,
CURLOPT_LOGIN_OPTIONS,CURLOPT_INTERFACE,CURLOPT_HEADEROPT,
CURLOPT_DNS_INTERFACE,CURLOPT_DNS_LOCAL_IP4,CURLOPT_DNS_LOCAL_IP6,
CURLOPT_EXPECT_100_TIMEOUT_MS,CURLOPT_MAIL_AUTH,CURLOPT_PROXY_SERVICE_NAME,
CURLOPT_TCP_KEEPALIVE,CURLOPT_TCP_KEEPIDLE,CURLOPT_TCP_KEEPINTVL,
CURLOPT_TCP_NODELAY,CURLOPT_SSL_VERIFYSTATUS,**

Visitez: http://curl.haxx.se/ pour une explication plus fine de chacune de ces options.

- Pages WEB.

Il existe deux façons de charger une page WEB, soit via une fonction de rappel soit dans un fichier.

Rappel

La première possibilité consiste à associer via l'opérateur « avec » un objet curl avec une fonction de rappel dont la signature est la suivante :

fonction url_rappel(chaine contenu,myobjet o);

La fonction sera associée de la façon suivante:

url u(o) avec url_rappel.

Dans ce cas, il faut utiliser la méthode *url(chaine).*

Exemple:

```
importe('kiffcurl');

fonction fonc(chaine contenu,omni o) {
        afficheligne(contenu);
}

curl c avec fonc;
//nous initialisons un proxy, à travers lequel nos pages WEB seront chargées.
c.proxy("http://myproxy.mycompany:5050");
//nous chargeons notre page web. Pour chaque bloc, func sera appelée...
c.url("http://www.liberation.fr/");
```

Fichier

L'autre possibilité consiste à utiliser la seconde méthode qui fournit un nom de fichier dans lequel sera sauvegardé le contenu de la page.

Exemple:

```
importe('kiffcurl');

curl c;
c.proxy("http://myproxy.mycompany:5050");
//Chargement de la page…
c.url("http://www.liberation.fr/","c:\temp\myfichier.html");
```

Exemple :

```
//Cette exemple montre comment l'on peut récupérer des informations provenant d'un moteur de recherché quelconque. (L'url utilisée ne correspond à aucun site existant à l'heure de rédaction de ce manuel)

chaine montexte;
fonction requêteur(chaine s,omni e) {
    montexte +=s;
}

curl appel avec requêteur;
//Le proxy
appel.proxy("my.proxy.com:8080");
//Quelques options fournis sous la forme de chaines de caractère…
appel.options("CURLOPT_HEADER", 0);
appel.options("CURLOPT_VERBOSE", 0);
appel.options("CURLOPT_AUTOREFERER",1);
appel.options("CURLOPT_FOLLOWLOCATION",1);
appel.options("CURLOPT_COOKIEFILE","");
appel.options("CURLOPT_COOKIEJAR","");
appel.options("CURLOPT_USERAGENT", "Mozilla/4.0 (compatible;)");

fonction larequête(tablechaines mots) {
    //Nous construisons notre requête
    chaine req="http://my.any.search.engine.com/html/?q=";
    montexte="";
    chaine lareq=req+mots.joins("+");
    appel.url(lareq);
    affficheligne(montexte);
}

larequête (["test","mot"]);
```