

CONTENTS

FOREWORD **6**

1 Precautions	6
2 Scope	6
3 History	6
4 Acknowledgments	7

GETTING STARTED **8**

1 Introduction	8
2 Installation	8
2.1 Linux	8
2.2 Mac OS 9	9
2.3 Mac OS X	9
2.4 Solaris	10
2.5 Windows	10
2.6 Key Directory	11
3 hello, world	13
3.1 Files	13
3.2 Build	16
3.3 Execute	16
4 To be continued...	17

KEY LANGUAGE **18**

1 Introduction	18
1.1 Overview	18
1.2 Notation	18
1.3 Comments	18
1.4 Identifier	19
1.5 Keywords	19
1.6 Symbols	19
2 Modules	19
2.1 Overview	19
2.2 Module	20
2.3 Imports	20
2.4 Exports	20
2.5 Includes	21
2.6 File	21
2.7 Globals	21
2.8 Entities	21
2.9 Option	21

3 Classes	22
3.1 Overview	22
3.2 Class	22
3.3 Parents	22
3.4 Features	23
3.5 Inheritance	23
4 Objects	24
4.1 Overview	24
4.2 Object	24
4.3 Instance	25
4.4 Array	25
4.5 Record	25
5 Programs	26
5.1 Overview	26
5.2 Program	26
5.3 Machine	27
5.4 Futures	27
5.5 Storage	27
6 Routines and Exceptions	28
6.1 Routine	28
6.2 Exceptions	28
6.3 Arguments	29
6.4 Assertions	29
6.5 Locals	30
7 Instructions and Expressions	30
7.1 Overview	30
7.2 Instructions	30
7.3 Expression	30
7.4 Access	31
7.5 Assign	31
7.6 Call	32
7.7 Check	32
7.8 Conform	33
7.9 Create	33
7.10 Fetch	34
7.11 Get	34
7.12 Loop	34
7.13 Put	35
7.14 Recall	35
7.15 Redo	36
7.16 Run	36
7.17 Select	37
7.18 Share	37
7.19 Store	37
7.20 Supply	37
7.21 Parameters	38
7.22 Precedence	38
8 Constants and Literals	39
8.1 Constant	39

8.2 Binary and Unary Operators	39
8.3 Literals	40
8.4 Simple literals	40
8.5 Vector literals	41
8.6 String	41

LIBKEY

42

1 Introduction	42
2 Base Classes	42
2.1 class DUMMY	42
2.2 class ANY	42
3 Simple Classes	43
3.1 class NONE	43
3.2 class BOOLEAN	43
3.3 class CHARACTER	44
3.4 class INTEGER	45
3.5 class RATIONAL	47
3.6 class REAL	47
4 Vector Classes	48
4.1 class VECTOR	48
4.2 class BOOLEANS	51
4.3 class CHARACTERS	51
4.4 class INTEGERS	51
4.5 class RATIONALS	51
4.6 class REALS	52
5 String Class	52
5.1 Overview	52
5.2 class STRING	52
6 Arrays and Collections	54
6.1 class ARRAY	54
6.2 class COLLECTION	56
7 Data Classes	59
7.1 Overview	59
7.2 class HANDLE	59
7.3 class POINTER	59
8 Language Classes	60
8.1 class TYPE	60
8.2 class MESSAGE	60
8.3 class MACHINE	61
8.4 class FUTURE	61
8.5 class GENERATOR	61
9 Utilities	63
9.1 object DEBUGGER	63
9.2 object KERNEL	63
9.3 object MEMORY	64

KEY IN C**65**

1 Introduction	65
2 Files	65
2.1 Platform Header File	65
2.2 Key Header File	65
2.3 Module Header File	65
2.4 Module C Source File	66
2.5 Routine C Source Files	66
2.6 Main C Source File	67
3 Routines and Exceptions	68
3.1 Routine	68
3.2 Self, Result and Program	68
3.3 Arguments	69
3.4 Locals	69
3.5 Exceptions	70
3.6 Errors	71
3.7 Debugging	72
3.8 C Call	72
3.9 C Callback	72
4 Constructs	73
4.1 Overview	73
4.2 keyCall*	73
4.3 keyConform	74
4.4 keyCreate	74
4.5 keyFetch	74
4.6 keyGet	75
4.7 keyPut	75
4.8 keyRecall*	75
4.9 keyRun*	76
4.10 keyShare	77
4.11 keyStore	77
4.12 keySupply	77
5 Conversions	78
5.1 Overview	78
5.2 Void	78
5.3 Boolean	78
5.4 Character	79
5.5 Integer	79
5.6 Rational	79
5.7 Real	80
5.8 Handle	80
5.9 Pointer	81
5.10 Type	81
5.11 Message	81
5.12 Machine	82
5.13 Vector Address	82
6 Strings	83
6.1 Overview	83

CONTENTS

6.2 From C To Key	83
6.3 From Key To C	84
6.4 From Key To C size	84
6.5 String Address	85
7 Resources	85

KEYSBUG **86**

1 Introduction	86
2 To be continued...	86

FOREWORD

1 Precautions

I worked on the Key language for almost ten years. I developed a new version every three years or so. Meanwhile, I used it to build applications. But this is my first attempt to document it myself.

In the past, the language was bound to authoring tools and more or less explained by their publishers. I met teams and users from time to time to present Key, and replied to their questions on discussion lists.

Today, the language is freely available and can be used to build whatever you want. It becomes necessary to have a kind of guide to Key, separated from the documentation of applications.

I expected it would be much tougher to write a small book than a big software. And it was. And it still is... So please consider the book as a work in progress. I am sure it will benefit a lot from your feedback.

Also, as you probably already noticed, my native language is French!

2 Scope

The title of the book tells you what it is. It is a collection of notes about the Key language. The book assumes you are more or less familiar with object and thread programming. If you want to use Key in C, it also assumes you know the C language.

The first chapter introduces you to the Key language, Libkey, Key in C and Keysbug: how to install them, how to use them.

The second chapter presents the Key language itself, its syntactical and lexical structure and its main concepts.

The third chapter comments the classes and objects of Libkey, the runtime library of the Key language.

The fourth chapter annotates the macros and functions of Key in C, the C application programming interface of the Key language and Libkey.

The fifth chapter describes Keysbug, the interactive debugger of the Key language.

It probably makes sense to walk through the book in that order once. But it is also intended to be used as a reference when you will be reading or writing Key and Key in C source files.

3 History

The Key language was designed for Arborescence, a CD-ROM publisher. The language was part of the custom authoring tool that Arborescence used to produce tens of titles. Since then, Key has been used directly or indirectly to develop thousands of applications.

In 1993, Apple Computer published the tool and named it the Apple Media Tool. The Key language was called the Apple Media Language (Key 1). In 1996, Arborescence developed and Apple Computer published the second version of the Apple Media Tool and the Apple Media Language (Key 2).

In 1999 and 2000, Tribeworks published the first and second versions of iShell, an authoring tool for kiosk, CD-ROM and client-side applications. The iShell Editor and Runtime are based on a version of the Key language developed from scratch in 1998 (Key 3).

For more than a year now, I am working with Artful to adapt the language to the development of server-side applications on multiple platforms. Three months ago, Artful and Tribeworks announced Webkool and Key 4, both available with an open source license.

4 Acknowledgments

Put the blame for the Key language on me... But it would be worse without the help of many people. Let me cite a few, with apologies to the others.

A significant role was played by Alain Soquet who designed the language with me. It is our experiments with C and Lisp dialects and our discussions on the merits of Eiffel, Oberon, Self, Smalltalk, and so on, that made Key what it is.

Key would not exist today without the trust of my partners, Max Derhy, Duncan Kennedy and Olivier Beudet, who, with Arborescence, Tribeworks or Artful, took the risk to market products or services based on an obscure and radical language.

I am particularly grateful to my colleagues, Gilbert Amar, Sebastien Burel, Marc Van Olmen and Mark Wharton, who, at Arborescence, Tribeworks and Artful, were the first to program with Key 1, 2, 3 and 4. Their suggestions and tests were essential to enhance and tune the language.

Last but not least I would like to thank the teams and the users of Apple Media Tool, iShell and Webkool for their support and enthusiasm, and to thank especially the people who dared to look under the hood to develop their project with the Key language.

Patrick Soquet
October 2001

GETTING STARTED

1 Introduction

The Key tool, the compiler and linker of the Key language and Libkey, the runtime library of the Key language are currently available on Linux (Pentium and PowerPC), MacOS 9 (PowerPC), Mac OS X (PowerPC), Solaris 7 or 8 (Sparc) and Windows 98, ME, NT4 or 2000 (Pentium).

Keysbug, the interactive debugger of the Key language is currently available on MacOS 9, Mac OS X and Windows 98, ME, NT4 or 2000. It can be used remotely to debug other platforms.

All the sources are available through anonymous cvs. The cvs root is

```
:pserver:cvs-pub@cvs.webkool.net:/export/home/cvs
```

The cvs password is

```
freeisgood
```

2 Installation

2.1 Linux

This installation assumes that your shell is tcsh.

ENVIRONMENT

Prepare your environment by adding these lines to your ~/.tcshrc file:

```
setenv CVSROOT :pserver:cvs-pub@cvs.webkool.net:/export/home/cvs
setenv KEY_HOME ~/dev/key
#setenv KEYSBUG_HOST 10.0.0.5:5001 # address of a machine with Keysbug
setenv PATH ${PATH}:${KEY_HOME}/bin
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${KEY_HOME}/lib/debug
```

CHECKOUT

Create the dev directory and go inside:

```
cd ~
mkdir dev
cd dev
```

Checkout the key repository:

```
cvs login
cvs checkout key
cvs logout
```

BUILD

Build the Key tool and Libkey:

```
cd ~/dev/key/Linux
make
```

2.2 Mac OS 9

You need MacCVS 3.1 and the MPW 3.5 to checkout and build Key.
 You can get MacCVS 3.1 at <<http://www.winevs.org/download.html>>

ENVIRONMENT

Prepare your environment by creating a MPW script in "Macintosh HD:MPW:Startup Items" with these lines:

```
Set -e KEY_HOME "Macintosh HD:dev:key:"
Set -e Commands "{KEY_HOME}bin:" ; {Commands}
```

CHECKOUT

- Use the Finder to create the "Macintosh HD:dev" folder
- Launch MacCVS
- In the CVS Admin menu, select Preferences..., enter the cvs root and select Password (pserver) in the Authentication pane.
- In the CVS Admin menu, select Login..., enter the cvs password
- In the CVS Admin menu, select Checkout module..., choose the "Macintosh HD:dev" folder, enter key as the module name
- Click OK and wait...
- Quit MacCVS

BUILD

To build the Key tool and Libkey, launch the MPW, select the Worksheet and enter:

```
setdirectory "{KEY_HOME}MacOS"
build.all
```

2.3 Mac OS X

You need the Developer Tools and the Terminal to checkout and build Key.
 This installation assumes that your shell is tcsh.

ENVIRONMENT

Prepare your environment by adding these lines to your ~/.login file:

```
setenv CVSROOT :pserver:cvs-pub@cvs.webkool.net:/export/home/cvs
setenv KEY_HOME ~/dev/key
setenv PATH ${PATH}:${KEY_HOME}/bin
```

CHECKOUT

Create the dev directory and go inside:

```
cd ~
mkdir dev
cd dev
```

Checkout the key repository

```

cvs login
cvs checkout key
cvs logout

```

BUILD

Build the Key tool and Libkey:

```

cd ~/dev/key/MacOSX
make

```

2.4 Solaris

This installation assumes that your shell is tcsh.

ENVIRONMENT

Prepare your environment by adding these lines to your ~/.tcshrc file:

```

setenv CVSRROOT :pserver:cvs-pub@cvs.webkool.net:/export/home/cvs
setenv KEY_HOME ~/dev/key
#setenv KEYSBUG_HOST 10.0.0.5:5001 # address of a machine with Keysbug
setenv PATH ${PATH}:${KEY_HOME}/bin
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${KEY_HOME}/lib/debug

```

CHECKOUT

Create the dev directory and go inside:

```

cd ~
mkdir dev
cd dev

```

Checkout the key repository:

```

cvs login
cvs checkout key
cvs logout

```

BUILD

Build the Key tool and Libkey:

```

cd ~/dev/key/Solaris
make

```

2.5 Windows

You need WinCVS 1.2 and MSVC++ 6.0 to build key.
You can get WinCVS 1.2 at <<http://www.wincvs.org/download.html>>

ENVIRONMENT

On Windows 98 and Me, add these lines to your autoexec.bat:

```

SET KEY_HOME=C:\dev\key
SET KEYSBUG_HOST=localhost:5001

```

```
SET MSDEV=C:\Program Files\Microsoft Visual Studio\VC98
SET PATH=C:\WINDOWS;C:\WINDOWS\COMMAND;C:\dev\key\bin
```

On Windows NT4 and 2000:

- Right-click My Computer and Select Properties to get the System Properties dialog box
- Select the Advanced tab
- Click the Environment Variables... button to get the Environment Variables dialog box
- In the User Variables pane click the New... button to get the New User Variable dialog box
- Enter KEY_HOME as the name, C:\dev\key as the value and click OK.
- In the User Variables pane click the New... button to get the New User Variable dialog box
- Enter KEYSBUG_HOST as the name, localhost:5001 as the value and click OK.
- In the User Variables pane click the New... button to get the New User Variable dialog box
- Enter MSDEV as the name, C:\Program Files\Microsoft Visual Studio\VC98 as the value and click OK
- In the System Variables pane, select the PATH variable and click the Edit button
- Append ;C:\dev\key\bin to the value and click OK
- Click OK to close the Environment Variables dialog box
- Click OK to close the System Properties dialog box

CHECKOUT

- Use the Explorer to create the C:\dev folder
- Launch WinCVS
- In the Admin menu, select Preferences..., enter the CVS root and select "passwd" file on the CVS server in the Authentication pane.
- In the Admin menu, select Login..., enter the CVS password
- In the Create menu, select Checkout Module..., enter key as the module name and C:\dev as the local folder
- Quit WinCVS

BUILD

To build the Key tool and Libkey, launch the MS-DOS or Command Prompt and enter:

```
C:
cd dev\key\Win32
build.bat
```

2.6 Key Directory

When the Key directory is installed, it contains the following directories:

bin

The bin directory contains mkkey and the Key tool that you will use to build Key softwares. On Mac OS, Mac OS X and Windows, it also contains Keysbug.

includes

The includes directory contains the C headers of Libkey. The C application programming interface of the Key language is annotated in the fifth chapter, Key In C.

KERNEL

The KERNEL directory contains the Key sources of Libkey. The classes and objects of the KERNEL are commented in the fourth chapter, Libkey.

lib

The lib directory contains the debug and release version of Libkey that you will use to run Key softwares.

Metrowerks

The Metrowerks directory contains projects to build the CodeWarrior plug-ins. A compiler, a preferences panel and a settings panel are provided.

PCCTS

The PCCTS directory contains the PCCTS headers and tools used to transform the grammar of the Key language into C sources. See <<http://www.antlr.org>>.

resources

The resources directory contains the cursors and icons of Keysbug for Mac OS, Mac OS X and Windows.

sources

The sources directory contains the C sources of the Key tool, Libkey and Keysbug.

tmp

The tmp directory contains the intermediate files generated by the compilers and linkers to build Key.

Linux**MacOS****MacOSX****Solaris****Win32**

The Linux, MacOS, MacOSX, Solaris and Win32 directories contain the make files to build Key on the corresponding platform. They also contain fragments of make files that will be combined by the Key tool to generate make files for Key softwares: debug.mk, release.mk, main.mk and module.mk. Look at such files to know the default options for compilers and linkers.

3 hello, world

Let us start with the usual example.

3.1 Files

Here are the directories and files to create...

```

dev
  key
  tutorial          # Project directory
    hello          # Module directory
      hello.kh     # Module header
      hello.km     # Module source
      hello.mk     # Module make fragments
      hello.plist  # Mac OS X only
      hello.r      # Mac OS only
      hello.rc     # Windows only
    sources
      main.c
      main.k

```

Of course the overhead for such a trivial application is kind of funny! But the purpose is just to explain how to build and execute Key softwares and to provide a template to copy and paste.

Key softwares are made of modules. One Key software has at least one module, the application a.k.a. the main module. Usually Key softwares have many modules: libraries, plug-ins, scripts...

A set of related modules is a project. Each project has its own directory besides the Key directory. Inside the project directory, each module has its own directory which contains the modules files and the sources directory.

hello.kh

hello.kh is the module header that will allow your C code to use Key classes, objects and programs. Here it is an empty file.

hello.km

```

module is pHello
import
  KERNEL;
include
  main;
end;

```

hello.km is the module source. A main module has to reference a program. It is the program that will be run when the module will be executed. Here it is pHello.

hello.mk

```
<Linux>
KEY_OPTIONS = -m
</Linux>

<MacOS>
CREATOR = 'MPS '
TYPE = 'MPST'
KEY_OPTIONS = -m
LIBRARIES = "{PPCLibraries}PPCToolLibs.o"
</MacOS>

<MacOSX>
TYPE = 'APPL'
KEY_OPTIONS = -m
LIBRARIES = -framework CoreServices -framework Carbon
</MacOSX>

<Solaris>
KEY_OPTIONS = -m
</Solaris>

<Win32>
C_OPTIONS = /D _CONSOLE
KEY_OPTIONS = -m
LINK_OPTIONS = /subsystem:console
</Win32>
```

hello.mk is the module make. It contains compiler and linker options for each platform. The -m option is to tell the Key tool that hello is a main module.

Depending on the options and the platform, the Key tool will combine one part of the module make with two fragments of make file in the Linux, MacOS, MacOSX, Solaris and Win32 directories in the Key directory to generate a make file for your module.

hello.plist

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist SYSTEM
"file://localhost/System/Library/DTDs/PropertyList.dtd">
<plist version="0.9">
<dict>
  <key>CFBundleInfoDictionaryVersion</key>
  <string>6.0</string>
  <key>CFBundleDevelopmentRegion</key>
  <string>English</string>
  <key>CFBundleExecutable</key>
  <string>hello</string>
  <key>CFBundleIdentifier</key>
  <string>net.webkool.hello</string>
  <key>CFBundleVersion</key>
  <string>0.0.1d1</string>
  <key>CFBundlePackageType</key>
  <string>APPL</string>
  <key>CFBundleSignature</key>
  <string>????</string>
</dict>
</plist>

```

hello.plist is required only on Mac OS X. On Mac OS X, a module is a bundle and Libkey looks for the Key code in the bundle.

hello.r

```
read 'KEY4' (1) "hello.key";
```

hello.r is required only on Mac OS. On Mac OS, Libkey looks for the Key code in the resources of the module.

hello.rc

```
1 KEY4 DISCARDABLE hello.key
```

hello.rc is required only on Windows. On Windows, Libkey looks for the Key code in the resources of the module.

main.c

```

#include "system.h"
#include "key.h"
#include "runKey.kh"

int main(int argc, char* argv[])
{
    return keyRunMain(2048*1024, 1024*1024, 512*1024, 256*1024, 16*1024,
        "Hello",
#ifdef mkWin32
        GetModuleHandle(NULL),
#elif mkMacOS
        (void*)CurResFile(),
#elif mkMacOSX
        CFBundleGetMainBundle(),
#else
        argv[0],
#endif
        keyMain,
        argc,
        argv);
}

```

A main module requires a C main. `keyRunMain` will initialize Libkey, run the program referenced by the module and terminate Libkey.

main.k

```

program pHello(...)
do
    DEBUGGER.Print("hello, world");
end;

```

`pHello` is the program referenced by the module. It will be run when the module is executed.

3.2 Build

Change the directory to the hello directory and enter

```
mkkey -d hello
```

The `-d` option is to tell the `mkkey` script to build a debug version of the module.

The `mkkey` script executes the `Key` tool to generate a make file, then it executes the `make` command to process the make file. When it is necessary, the `Key` tool, the C compiler, the linker, and so on, are executed to build the module.

3.3 Execute

To execute hello, enter:

LINUX

```
cd ~/dev/tutorial/bin/debug  
./hello
```

MAC OS

```
directory "Macintosh HD:dev:tutorial:bin:debug"  
hello
```

MAC OS X

```
cd ~/dev/tutorial/bin/debug  
./hello.app/Contents/MacOS/hello
```

SOLARIS

```
cd ~/dev/tutorial/bin/debug  
./hello
```

WINDOWS

```
C:  
cd dev\tutorial\bin\debug  
hello.exe
```

It prints `hello, world` in Keysbug if it is available, or in your terminal.

4 To be continued...

KEY LANGUAGE

1 Introduction

1.1 Overview

This chapter presents the Key language itself, its syntactical and lexical structure and its main concepts. The order of the presentation is top-down: constructs are introduced from the largest to the smallest.

1.2 Notation

Any part of a Key source file corresponds to a syntactical or lexical structure known as a construct. Every construct is either a terminal or a non-terminal.

A terminal is a construct that stands for itself. Terminal includes keywords and symbols. A non-terminal is a construct that is defined in terms of other constructs by a production like

```
Void ::= `void`
```

Every non-terminal appears on the left-hand side of one and only one production. The right-hand side of a production uses the following convention:

a b	a followed by b
a b	a or b but not both
a?	a or nothing
a+	a one or more times
a*	a zero or more times
[a-z]	any character in the range indicated
[^abc]	any character not among the characters given
{a}	the non-terminal a
`a`	the terminal a

Parenthesis are used to group constructs.

Key is case insensitive so range and sequence of characters are presented without capital letters.

1.3 Comments

A Key comment begins with the double dash symbol and extends to the end of a line. Comments have no effect on the semantic and can appear at the end of any line.

EXAMPLE

```
-- this is a comment!
```

1.4 Identifier

A Key identifier begins with a letter and continues with zero or more letter, digit, underscore or question mark. An identifier is valid only and only if it is not a keyword.

Identifier ::= [a-z][a-z0-9_?]*

The Identifier construct is a lexical construct, no space, tabs or returns are allowed in its production.

1.5 Keywords

and	argument	binary	check	class	debug	do
else	end	export	external	false	from	has
has?	if	import	include	is	is?	loop
module	new	not	object	or	program	redo
release	result	self	share	step	then	true
unary	undo	until	use	void	while	with
xor						

1.6 Symbols

#	%	&	*	+	,	-
.	...	/	:	:=	;	<
<=	<>	=	>	>=	@	^
	~	!				

2 Modules

2.1 Overview

The architecture of Key software is based on units called modules.

What you build with the Key language is always a module. The Key tool and Libkey compile and link only modules. You can make and use modules as applications, libraries, plug-ins or scripts but, from the point of view of the Key language, they are the same.

Libkey itself is a module, a library that Key software always uses. Usually Key software will be delivered as an application with its required libraries and its optional plug-ins or scripts. At runtime, Libkey is in charge of loading and linking modules.

2.2 Module

```
.km ::= {Module} `;` {Entities}
.k ::= { Entities }
Module ::= `module` `is` {Identifier}
          ({Imports})?
          ({Exports})?
          ({Includes})?
          ({Globals})?
          `end`
```

A module consists of one or more source files: the module source file itself (.km) and the exported or included source files (.k). The module source file has to begin with the definition of the module and can continue with the definition of entities like classes, objects or programs. The exported or included source files can only contain the definition of entities.

There is no constrain on where to define such or such entity. When you build a module, it is like if all the exported or included source files were appended to the module source file.

A module is also a constant. Libkey will use the constant to launch the application, or will return the constant when a plug-in or a script is loaded. Whatever is loading the plug-in or the script has then to check the returned value.

2.3 Imports

```
Imports ::= `import` (({File} | ({Option} {Files} `end`)) `;`)*
```

The list of imported files is the list of modules that are required by the module. A module never contains another module, it just contains a list of its required modules.

Prior to link a module, both the Key tool and Libkey incrementally and recursively load and link its required modules. No module is loaded twice and no modules can require themselves mutually, directly or indirectly.

To load a module named foo, the Key tool looks for a file named foo.key in its input directories; depending on the platform, Libkey looks also for a file named foo.key or for the key resource of a file named foo, foo.dll or foo.bundle in the application directory and in the modules directory of the application directory.

2.4 Exports

```
Exports ::= `export` (({File} | ({Option} {Files} `end`)) `;`)*
```

The list of exported files is the list of source files that describe the public entities of the module. Public entities are visible to modules that will import the module.

All exported files are parsed when the module is compiled. To parse an exported file named foo, the Key tool looks for a file named foo.k in its input directories.

Entities described in the module file itself are always exported.

2.5 Includes

```
Includes ::= 'include' (({File} | ({Option} {Files} 'end')) ';'*)
```

The list of included files is the list of source files that describe the private entities of the module. Private entities are invisible to modules that will import the module.

All included files are parsed when the module is compiled. To parse an included file named foo, the Key tool looks for a file named foo.k in its input directories.

2.6 File

```
File ::= {Identifier}
```

If your file system is case sensitive, be aware that the capitalization of the name of the files in the Imports, Exports and Includes constructs is significant.

2.7 Globals

```
Globals ::= 'use' (({Global} | ({Option} {Includes} 'end')) ';'*)
Global ::= {Identifier}
```

The Globals construct declares the global variables that the Module will use. Globals can be conditionally compiled depending on options.

Initially all global variables contain void and are shared. When a thread accesses a global variable, it owns the global variable. If another thread wants to access the global variable, it waits until the global variable is shared again.

To share a global variable, a program uses the Share construct. If a global variable is used only by a single thread, it never has to be shared.

2.8 Entities

```
Entities ::= (({Entity} | ({Option} {Entities} 'end')) ';'*)
Entity ::= {Class} | {Object} | {Program}
```

Entities are the building blocks of Key software. There are three kind of entities in Key: classes, objects and programs. Entities can be conditionally compiled depending on options.

The definition of a entity will define one or more constants.

2.9 Option

```
Option ::= 'debug' | ('release' {Identifier} ':')
```

Files, globals, entities, parents, features, properties, locals and instructions can be conditionally compiled depending on options.

The debug option is on when you build the debug version of a module, i.e. when you use the `-d` option in the Key tool command line. The release options are on if their identifier matches the argument of one of the `-r` options in the Key tool command line.

In Libkey, the debug option is on if you use the debug version of Libkey. You can set the release options thanks to methods of the KERNEL object. Options will only condition the way scripts are compiled.

EXAMPLE

```
module foo is pFoo
import
  KERNEL;
export
  Foo;
  release Editor:
    IOFoo;
end;
end;
```

3 Classes

3.1 Overview

Classes are one of the three building blocks of Key software. A class defines the structure and the behavior of a set of values. Such values are the instances of the class.

A class can define a structure and a behavior from scratch but often a class inherits from other classes to refine the definition of their structure and their behavior.

3.2 Class

```
Class ::= `class` {Identifier} (`is` {Parents})? (`has` {Features})? `end`
```

Every class has a name and a text which describes its parents and its features. The name of a class is its identifier. The parents of a class define the inheritance of the class. The features of a class define or refine the structure and the behavior of its instances.

The definition of a class corresponds to the definition of a TYPE constant.

3.3 Parents

```
Parents ::= (({Parent} | ({Option} {Parents} `end`)) `;`)*
Parent ::= {Constant}
```

The Parents define the inheritance of a class. Parents can be conditionally compiled depending on options.

A class can have multiple parents but cannot be its own parent, directly or indirectly.

A parent has to be a TYPE Constant.

3.4 Features

```
Features ::= (({Feature} | ({Option} {Features} `end`)) `;`)*
Feature ::= {Field} | {Method}
Field ::= {Identifier}
Method ::= ({Identifier} | {Operator}) {Routine}
```

The features of a class are its fields and its methods. Features can be conditionally compiled depending on options.

The fields of a class define or refine the structure of its instances. Fields are containers where you can get or put values.

The methods of a class define or refine the behavior of its instances. Methods are routines that you can call or recall.

The definition of a feature corresponds to the definition of a MESSAGE constant.

3.5 Inheritance

A class inherits the fields and the methods of its parents. It means that a class can refine the structure and the behavior defined by its parents only by adding features, never by subtracting features.

A class can also refine the behavior defined by its parents by overriding methods. You override the method of a parent when you define a method with the same identifier but a different routine.

Fields are just containers, there is nothing to refine and you cannot override them.

Multiple inheritance can cause conflicts if parents define features with the same name.

Parents are recursively inherited but are inherited once and only once so there is no conflict because of shared parents.

To solve conflicts between methods, you have to override the method, except if one of the conflicting method is already overriding the other, because Key defaults to the most refined method.

EXAMPLE

```
class A has Foo; end;
class B has Foo; end;
class C has Foo() do end; end;
class D has Foo() do end; end;
class E is A; end;
class F is A; end;
class G is C; end;
class H is C; end;
```

```

class I is C; has Foo() do end; end;
class J is C; has Foo() do end; end;

class AB is A; B; end; -- Error: AB has two Foo fields
class AC is A; C; end; -- Error: AC has one Foo field and one Foo method
class CD is C; D; end; -- Error: AB has two Foo methods
class EF is E; F; end; -- EF has one Foo field defined by A
class GH is G; H; end; -- GH has one Foo method defined by C
class GI is G; I; end; -- GI has one Foo method defined by I
class IJ is I; J;
has
  Foo() -- Override the Foo method to solve the conflict
  do
    self.I:Foo(); -- recall the Foo method of I
    self.J:Foo(); -- recall the Foo method of J
  end;
end;

```

4 Objects

4.1 Overview

Objects are one of the three building blocks of Key software.

Firstly objects are used for input and output. Key is designed to be used with tools which parse and generate source files. The object construct allows such tools to exchange data.

Secondly, objects are used for templates. If you look at Key software, you will notice that few values are created by instantiating a class like:

```
result := new cFoo;
```

Instead, most values are created by duplicating an object like:

```
result := oFoo.Dub();
```

Thirdly, objects are used for filters. Even if the value of an object does not change during the execution of Key software, the behavior of an object can be used to transform values.

4.2 Object

```
Object ::= 'object' {Identifier} 'is' {Instance}
```

Every object has a name and a text which describes its value. The name of an object is its identifier. The value of an object is, like all values in Key, the instance of a class.

There are constructs to recursively describe instances of all classes, and to refine classes in the description of instances themselves.

The definition of an object correspond to the definition of a constant. The value of an object does not change during the execution of Key software and is shared by all threads.

4.3 Instance

```
Instance ::= {Constant} | {Literal} | {Array} | {Record}
```

The Instance construct is the description of the value of the object.

The Constant construct allows you to put a forward or a backward reference in the definition of an object. Objects can refer to themselves, directly or indirectly.

The Literal construct defines instances of the simple classes, the vector classes and the STRING class.

The Array construct defines instances of the ARRAY class.

The Record construct defines instances of all the other classes.

EXAMPLE

```
object foo is 1:2;
object foo is { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
object foo is "foo";
```

4.4 Array

```
Array ::= `[` ({Instance} (`,` {Instance})*)? `]`
```

The Array construct define instances of the ARRAY class. It is a list of instances between brackets, separated by commas.

EXAMPLE

```
object foo is [ "", "", "" ];
object foo is [
    binary +,
    binary -,
    Compare
];
```

4.5 Record

```
Record ::= {Constant} (`has` {Features})? (`with` {Fields})? `end`
Fields ::= ((({Constant} `is` {Instance})
    | ({Option} {Fields} `end`)) `;`)*
```

The Record construct begins with a constant. It has to be a TYPE constant and it corresponds to the class of the instance being defined.

Then, optionally, you can change the features of the class: you can add fields or methods and override methods. It is equivalent to define an anonymous class whose only instance is the instance being defined.

The Record construct ends with the Fields construct. The Fields construct allows you to define the value of the fields of the instance.

The constants of the Fields construct have to be MESSAGE constants, and the class of the instance being defined has to have fields that matches the MESSAGE constant.

EXAMPLE

```
object oFoo is cFoo
with
  Foo is cFoo
  has
    Goo;
  with
    Foo is oFoo;
    Goo is 1;
  end;
end;
```

5 Programs

5.1 Overview

Programs is one of the three building blocks of Key software.

Key software can have multiple preemptive threads, i.e. multiple sequences of code to execute concurrently. Programs are used to define thread pools.

The system schedules the execution of threads without the collaboration of the application and can map separate threads to separate processors.

5.2 Program

```
Program ::= `program` {Identifier} {Routine}
```

Every program has a name and a text which describes its routine. The name of a program is its identifier. The routine of a program defines the sequence of code to execute when it runs.

The definition of a program corresponds to the definition of a MACHINE constant.

When you build an application, the module has to be a MACHINE constant bound to a program which will be the entry point of the main thread.

EXAMPLE

```

program pGetAt(theTarget, theIndex)
  do
    result := theTarget.GetAt(theIndex);
  end;

```

5.3 Machine

A MACHINE is no single thread, it is a thread pool, i.e. several threads that use the same sequence of code defined by the program. To run a program means to post a request to a thread pool. Requests are queued until a thread is available in the pool to handle it. The parameters are passed by value, they are cloned into the request.

The MACHINE class allows to control the thread pool. There are methods to get and set the heap size, the stack size and the pool size. Every thread has a separate heap and stack. The heap is used for all the instances created or cloned during the execution of the thread. It will grow if necessary.

The pool size is the number of threads that the MACHINE is allowed to create to handle its requests. By default it is one, meaning that the MACHINE handles one request at a time. The TestPoolSize method allows to tune the MACHINE. If the result is less than zero, it is the number of requests that are waiting a thread to be handled. If the result is more than zero, it is the number of threads that are waiting a request to handle.

5.4 Futures

A program that runs another program does not wait, it continues immediately its own sequence of code until it needs the value of the result of another program, i.e. it waits by necessity.

The result of a program is an instance of the class FUTURE. To test if the value of a FUTURE is available, call its Ready method. To get the value of a FUTURE, call its Evaluate method.

EXAMPLE

```

result := pGetAt(self, 1); -- does not wait
-- does something else
result := result.Evaluate(); -- waits until result is available

```

When a thread waits, it sleeps until it is awake by the availability of the FUTURE. Beware of mutually waiting threads!

5.5 Storage

Every thread can use the program keyword to access and assign an instance that is owned by the thread and is resident to the thread, i.e. such instance is not garbage collected when the thread exits.

EXAMPLE

```

program pCollection(theMessage, ...)
do
  if program = void then
    program := new COLLECTION;
  end;
  result := program.theMessage(...);
end;

pCollection(PutAt, "1", 1);
pCollection(PutAt, "2", 2);
result := pCollection(GetAt, 1); -- result = "1"

```

It is a way to define services without global variables. Usually the pool size of a machine that defines a service will be one, the default. If the machine has several threads, each thread will have its own storage.

6 Routines and Exceptions

6.1 Routine

```

Routine ::= `( ` {Arguments} ` ) `
          {Assertions}
          ( {External} | {Internal} )

External ::= `external` {String}

Internal ::= ( `use` {Locals} ) ?
            ( `do` {Instructions} ) ?
            ( `undo` {Instructions} ) ?
            `end`

```

The Routine construct defines the sequence of instructions to execute when a Program is run or when a Method is called or recalled and the sequence of instructions to execute when an exception is triggered.

There are two kind of routines, External and Internal.

External routines are written in C. The String is the name of the C function that will be executed. The Key in C chapter describes how to implement Key routines in C.

Internal routines are written in Key. Following the do keyword are the instructions to execute when a Program is run or when a Method is called or recalled. Following the undo keyword are the instructions to execute when an exception is triggered.

6.2 Exceptions

Exceptions are triggered by Libkey or external routines in case of errors.

When an exception is triggered by an instruction of the Do block, the remaining instructions of the Do block are not executed.

If there is an Undo block, the instructions of the Undo block are executed to the end or until a Redo instruction is executed. If a Redo instruction is executed, the Do block is executed again but the values contained in the arguments, local variables and result remain what they are.

If there is no Undo block or if no Redo instruction is executed, the routine has failed. If it is the routine of Method, an exception is triggered by the instruction that called or recalled the Method. If it is the routine of a Program, the thread that is executing the Program exits.

So an exception is processed by all routines in the chain of Call or Recall instructions until a Redo instruction is executed or until the thread exits.

EXAMPLE

```
Try(theCount, theMessage, ...)
  -- Attempt theCount times to call theMessage method
  theCount is? INTEGER;
  theCount > 0;
do
  self.theMessage(...);
undo
  theCount := theCount - 1;
  if theCount > 0 then
    redo;
  end;
end;
```

6.3 Arguments

```
Arguments ::= (({Argument} ', ')* ({Argument} | '...'))?
Argument ::= {Identifier}
```

A routine has zero or more arguments, the initial value of the arguments are the parameters that are passed by the Call, Recall or Run instruction.

The number of parameters can be bigger but cannot be smaller than the number of arguments. Otherwise the routine is not executed and an exception is triggered.

Routines always have a special argument identified by the self keyword. The value contained by self is the instance being called or recalled. If the routine is the routine of a program, self is void.

6.4 Assertions

```
Assertions ::= (({Assertion} | ({Option} {Assertions} 'end')) ';')*
Assertion ::= {Expression}
```

Assertions are expressions that state requirements that have to be true to execute the routine. Assertions can for instance verify the value, the type and the range of the arguments, verify the state of self, and so on...

Assertions are used to enhance the correctness and the reliability of Key software. Assertions are monitored by the debug version of libkey and trigger exceptions unless the result of their expression is true.

Note that assertions are outside of the Do block of the Routine and that the exceptions they trigger do not execute the Undo block of the Routine.

6.5 Locals

```
Locals ::= (({Local} | ({Option} {Locals} `end`)) `;`)*
Local ::= {Identifier}
```

The Locals construct declares the local variables that the Routine will use. The initial value of a local variable is void.

Routines always have a special local variable identified by the result keyword. The value contained by result will be returned by the Call, Recall or Run expression that executed the routine.

7 Instructions and Expressions

7.1 Overview

Instructions and Expressions are used to define the sequence of code a routine will execute.

Most instructions and expressions can fail and trigger an exception if they are misused. The description of each instruction or expression details when it would fail.

7.2 Instructions

```
Instructions ::= (({Instruction} | ({Option} {Instructions} `end`)) `;`)*
```

Instructions can be conditionally compiled depending on options.

```
Instruction ::= {Assign} | {Call} | {Check} | {Iterate} | {Put}
               | {Recall} | {Redo} | {Run} | {Select} | {Share} | {Store}
```

7.3 Expression

```
Expression ::= `(` {Expression} `)`
              | {Access} | {Call} | {Conform} | {Create} | {Fetch} | {Get}
              | {Recall} | {Retrieve} | {Run} | {Supply}
```

Expressions are constructs that return a value.

```

Boolean-Expression ::= {Expression}
Integer-Expression ::= {Expression}
Machine-Expression ::= {Expression}
Message-Expression ::= {Expression}
Type-Expression ::= {Expression}
Target-Expression ::= {Expression}

```

The production of some instructions and expressions use variants of the Expression construct.

The Boolean-Expression, Integer-Expression, Machine-Expression, Message-Expression or Type-Expression constructs are expressions that must return a value which is an instance of the BOOLEAN, INTEGER, MACHINE, MESSAGE or TYPE class.

The Target-Expression construct just emphasizes an expression that returns a value which is the target of the produced instruction or expression.

7.4 Access

```

Access ::= {Argument} | {Local} | 'program' | 'result' | 'self'
         | {Count} | {Etc}

```

The Access expression returns the value of an argument, a local variable, program, result or self.

```
Count ::= '#' 'argument'
```

The Count expression returns the number of arguments.

```
Etc ::= 'argument' '@' {Integer-Expression}
```

The Etc expression returns the value of the argument at the index returned by its Integer-Expression.

The Etc expression fails if:

- The Integer-Expression does not return an instance of the INTEGER class.
- The index is less than one or more than the number of arguments.

EXAMPLE

```

Foo(theArgument, ...)
  use
    aLocal;
  do
    result := theArgument; -- access Argument
    if #argument > 1 then -- access Count
      result := argument @ 2; -- access Etc
    end;
    result := aLocal; -- access Local
  end;

```

7.5 Assign

```
Assign ::= ({Argument} | {Local} | 'program' | 'result') ':=' {Expression}
```

The Assign instruction replaces the value of an argument, a local variable, program or result by the value of its Expression.

EXAMPLE

```
result := void;
```

7.6 Call

```
Call ::= ({Target-Expression} `.` {Message-Expression} `(` {Parameters} `)` )
        | {Expression} {Binary-Operator} {Expression}
        | {Unary-Operator} {Expression}
```

The Call expression executes the Routine of a Method of an Instance with zero or more parameters. The expression returns the result of the Routine.

The conventional binary and unary expressions are a special form of the Call expression.

Usually the Message-Expression is just a Fetch expression which returns a MESSAGE constant but it can be any expression that returns a MESSAGE constant.

The Call expression can also be used as an Instruction.

The Call expression fails if:

- The Message-Expression does not return a MESSAGE Constant.
- The class of the instance returned by the Target-Expression does not supply a Method that matches the MESSAGE Constant.
- The number of parameters is less than the number of arguments of the Routine of the Method.
- The Routine of the Method fails.

EXAMPLE

```
self.PutAt(void, 1);
result := self.GetAt(1);
self.(self.Method)();
result := result + 1; -- result := result.binary +(1);
result := -result; -- result := result.unary -();
```

7.7 Check

```
Check ::= `check` {Expression}
```

The Check instruction states a requirement that has to be true whenever the execution reaches the instructions.

Check instructions are monitored by the debug version of libkey and trigger exceptions unless the result of their expression is true.

EXAMPLE

```
check result <> void;
```

7.8 Conform

```
Conform ::= {Target-Expression} 'is?' {Type-Expression}
```

The Conform expression returns true if the TYPE constant returned by the Type-Expression is directly or indirectly a Parent of the class of the instance returned by the Target-Expression; else it returns false.

Usually the Type-Expression is just a Fetch expression which returns a TYPE constant but it can be any expression that returns a TYPE constant.

The Conform expression fails if:

-The Type-Expression does not return a TYPE Constant.

EXAMPLE

```
"foo" is? ARRAY -- return false
"foo" is? STRING -- return true
"foo" is? Clone -- fail
"foo" is? KERNEL.TypeOf("foo") -- return true
```

7.9 Create

```
Create ::= { Create-Literal } | { Create-Array } | {Create-Any}
```

The Create expressions create instances and return them. The Create-Literal expression creates an instances of the simple classes, the vector classes or the STRING class. The Create-Array expression creates an instance of the ARRAY class. The Create-Any expression creates an instance of the other classes.

```
Create-Literal ::= {Literal}
```

If you use a Literal in a Routine, you create an instance every time the expression is executed. Instances of the vector classes and the STRING class require memory in the heap of the thread that executes the expression.

The Create-Literal expression fails if:

-No memory can be allocated to the thread that executes the expression.

```
Create-Array ::= '[' {Parameters} ']'
```

The Create-Array expression use the Parameters construct like the Call and Recall expressions so you can create an ARRAY with the optional arguments.

The Create-Array expression fails if:

-No memory can be allocated to the thread that executes the expression.

```
Create-Any ::= 'new' {Type-Expression}
```

Usually the Type-Expression is just a Fetch expression that returns a TYPE constant but it can be any expression that returns a TYPE constant.

The Create-Any expression fails if:

-The Type-Expression does not return a TYPE Constant.

-No memory can be allocated into the thread that executes the expression.

EXAMPLE

```

result := "foo"; -- create a STRING
result := [ ... ]; -- create an ARRAY with the optional arguments
result := new COLLECTION.WithSize(2001); -- create a COLLECTION
result := new (KERNEL.TypeOf(self)); -- create an instance of self

```

7.10 Fetch

```
Fetch ::= {Constant} | {Global}
```

The Fetch expression returns the value of a Constant or a Global.

If the thread that executes the Fetch expression does not own the Global, it waits until the Global is shared to become the owner of the Global.

EXAMPLE

```

result := oFoo;
result := gFoo;

```

7.11 Get

```
Get ::= {Target-Expression} `.` {Message-Expression}
```

The Get expression returns the value of a Field of an Instance.

Usually the Message-Expression is just a Fetch expression which returns a MESSAGE constant but it can be any expression that returns a MESSAGE constant.

The Get expression fails if:

- The Message-Expression does not return a MESSAGE Constant,
- The class of the instance returned by the Target-Expression does not supply a Field that matches the MESSAGE Constant

EXAMPLE

```

result := self.Foo;
result := oFoo.(self.GetField());

```

7.12 Loop

```

Iterate ::= {Test-Loop} | {Loop-Test}
Test-Loop ::= {From} {Test} {Loop} {Step}? `end`
Loop-Test ::= {From} {Loop} {Step}? {Test}
From ::= `from` {Instructions}
Loop ::= `loop` {Instructions}
Step ::= `step` {Instructions}
Test ::= (`until` | `while`) {Boolean-Expression}

```

The Loop instruction provides iterative execution of instructions.

The Loop instruction fails if:

-The Boolean-Expression does not return false or true.

EXAMPLE

```

from
  result := 1;
while result <= 10 loop
  self.Foo(result);
step
  result := result + 1;
end;

from
  result := 10;
loop
  self.Foo(result);
step
  result := result - 1;
until result = 0;

```

7.13 Put

Put ::= {Target-Expression} `.` {Message-Expression} `:=` {Expression}

The Put instruction replaces the value of the Field of an Instance by the value of its Expression.

Usually the Message-Expression is just a Fetch expression which returns a MESSAGE constant but it can be any expression that returns a MESSAGE constant.

The Put instruction fails if:

-The Message-Expression does not return a MESSAGE Constant.

-The class of the instance returned by the Target-Expression does not supply a Field that matches the MESSAGE Constant

-The Target-Expression returns an instance that is a Constant.

EXAMPLE

```

self.Foo := void;
gFoo.(self.GetField()) := self;

```

7.14 Recall

Recall ::= {Target-Expression} `.` {Type-Expression} `:` {Message-Expression} `(` {Parameters} `)`

The Recall expression executes the Routine of a Method of a Parent of an Instance with zero or more parameters. The expression returns the result of the Routine.

Usually the Type-Expression is just a Fetch expression that returns a TYPE constant but it can be any expression that returns a TYPE constant.

Usually the Message-Expression is just a Fetch expression which returns a MESSAGE constant but it can be any expression that returns a MESSAGE constant.

The Recall expression can also be used as an instruction.

The Recall expression fails if:

- The Type-Expression does not return a TYPE Constant,
- The Message-Expression does not return a MESSAGE Constant,
- The class of the instance returned by the Target-Expression does not conform to a Parent that matches the TYPE Constant
- The class corresponding to the TYPE Constant does not supply a Method that matches the MESSAGE Constant
- The number of parameters is less than the number of arguments of the Routine of the Method.
- The Routine of the Method fails.

EXAMPLE

```
self.cFoo:Foo("Foo");

DoAsFoo(theMessage, ...)
do
    result := self.cFoo:theMessage(...);
end;
```

7.15 Redo

Redo ::= `redo`

The Redo instruction is used in the Undo block to repeat the execution of the Do block. The values of arguments, local variables and result do not change.

7.16 Run

Run ::= {Machine-Expression} `(` {Parameters} `)`

The Run expression executes the Routine of a Program with zero or more parameters. The expression returns the result of the Routine.

Usually the Machine-Expression is just a Fetch expression that returns a MACHINE constant but it can be any expression that returns a MACHINE constant.

The Run expression can also be used as an instruction.

The Run expression fails if:

- The Machine-Expression does not return a MACHINE Constant,
- The number of parameters is less than the number of arguments of the Routine of the Program.
- The Routine of the Program fails.

EXAMPLE

```
pWritePool("Foo");

result := (self.ReadPool)("Foo");
```

7.17 Select

```
Select ::= {If} {Then} ('else' {If} {Then})* ('else' {Instructions})? 'end'
If ::= 'if' {Boolean-Expression}
Then ::= 'then' {Instructions}
```

The Select instruction provides conditional executions of instructions.

The Select instruction fails if:

-The Boolean-Expression does not return false or true.

EXAMPLE

```
if result <= 0 then
    result := 0;
else if result >= 10 then
    result := 100;
else
    result := result * result;
end;
```

7.18 Share

```
Share ::= 'share' {Global}
```

The Share instruction shares the Global.

EXAMPLE

```
share gFoo;
```

7.19 Store

```
Store ::= {Global} ':=' {Expression}
```

The Store instruction replaces the value of a Global by the value of its Expression.

If the thread that executes the Store instruction does not own the Global, it waits until the Global is shared to become the owner of the Global.

EXAMPLE

```
gFoo := void;
```

7.20 Supply

```
Supply ::= {Target-Expression} 'has?' {Message-Expression}
```

The Supply expression returns true if the MESSAGE constant returned by the Message-Expression matches a Field or a Method of the class of the instance returned by the Target-Expression; else it returns false.

Usually the Message-Expression is just a Fetch expression which returns a MESSAGE constant but it can be any expression that returns a MESSAGE constant.

The Supply expression fails if:

-The Message-Expression does not return a MESSAGE Constant,

EXAMPLE

```
"foo" has? GetAt -- return false
"foo" has? Clone -- return true
"foo" has? STRING -- fail
```

7.21 Parameters

```
Parameters ::= (({Expression} \,\' )* ({Expression} | \'...\'))?
```

The Parameters construct is used in the Call, Create-Array, Recall and Run constructs. It is a list of expressions separated by commas.

If the Parameters construct ends with the triple dots, it means that the anonymous arguments received by the routine are passed as parameters of the Call, Create-Array, Recall or Run constructs.

7.22 Precedence

In the absence of parentheses, expressions are associated according to the precedence of their keyword or symbol. If the precedences are equal, expressions are associated from left to right.

Here are the precedences from the highest to the lowest:

```
Level 9  new
Level 8  :
Level 4  .
Level 6  is?
Level 5  has?
Level 4  # , ~ , not , -
Level 5  % , / , *
Level 6  - , +
Level 1  <> , >= , <= , > , < , =
Level 0  @ , ^ , xor , | , or , & , and
```

EXAMPLE

```
new cFoo.WithSize(1) -- (new cFoo).WithSize(1)
self.cFoo:Foo(1); -- self.(cFoo:Foo)(1);
self.Foo.Goo -- (self.Foo).Goo
-5 + 4 * 3 + 2 / 1; -- ((-5) + (4 * 2)) + (2 / 1)
```

8 Constants and Literals

8.1 Constant

```
Constant ::= (({Identifier} '!'?) {Identifier}) | {Operator}
Operator ::= ('binary' {Binary-Operator}) | ('unary' {Unary-Operator})
```

A Constant binds an Identifier and a value that does not change during the execution of a Key software.

When you define a Class, you implicitly define a TYPE Constant, its Identifier is the Identifier of the Class, its value is an instance of the class TYPE.

When you define a Feature, you implicitly define a MESSAGE Constant, its Identifier is the Identifier of the Feature, its value is an instance of the class MESSAGE.

When you define an Object, you explicitly define a Constant, its Identifier is the Identifier of the Object, its value is described by the Instance description of the Object.

When you define a Program, you implicitly define a MACHINE Constant, its Identifier is the Identifier of the Program, its value is an instance of the class MACHINE.

When you import several modules, a Constant could be defined by more than one module. To avoid the ambiguity, you have to precede the Identifier of the Constant by an exclamation mark and the Identifier of its module.

Key itself define a set of Constant which is the set of Operator. An Operator constant is represented by one of the binary or unary operators preceded by the keyword binary or unary, its value is an instance of the MESSAGE class.

The Constant construct is used in the Instance description and in the Fetch expression.

8.2 Binary and Unary Operators

```
Binary-Operator ::= 'and' | 'or' | 'xor' | '&' | '|' | '^'
                  | '@' | '=' | '<' | '>' | '<=' | '>=' | '<>'
                  | '+' | '-' | '*' | '/' | '%'
Unary-Operator ::= 'not' | '~' | '#' | '-'
```

There are two kinds of operators: binary and unary. Operators alone are used in the conventional binary and unary expressions. There are no implicit conversions from type to type.

Operators preceded by the binary or unary keyword are just MESSAGE Constant and you can use them everywhere you can use messages, for instance in Call or Recall expressions or in Method features.

EXAMPLE

```
result + 1 = result.binary +(1)
- result = result.unary -()
```

```
binary +(it)
    it is? INTEGER;
    external "IntegerAdd";
```

```
unary -()
  external "IntegerMinus";
```

8.3 Literals

```
Literal ::= {Simple-Literal} | {Vector-Literal} | {String}
```

A literal is the representation of a value which, like all values in Key, is the instance of a class. It is the representation that selects the class of the instance.

The classes of the instances represented by literals are defined in Libkey.

There is a Literal construct for all the simple values, the vector values and the STRING value.

8.4 Simple literals

```
Simple-Literal ::= {Boolean} | {Character} | {Integer} | {Rational} | {Real}
| {Void}
```

Simple literals are lexical constructs, no spaces, tabs or returns are allowed in their production.

```
Boolean ::= `false` | `true`
```

There are two Boolean literal values: false and true.

The class of a Boolean literal value is BOOLEAN.

```
Character ::= `` ([^`] | `\\` | `\'` | (`\$` [0-9a-f]+)) ``
```

A Character literal represents a single Unicode character. It is a character between single quote. The single quote and the backslash have to be escaped with a backslash.

Only ASCII characters can be represented directly, all other characters have to be represented by a backslash, a dollar and hexadecimal digits. The number is the UCS-4 code of the character.

The class of a Character literal value is CHARACTER.

```
Integer ::= (`-`? [0-9]+) | (`$` [0-9a-f]+)
```

An Integer literal represents a 32-bit signed integer value. They have two forms: decimal and hexadecimal. The sign of the hexadecimal form depends on the most significant bit of its representation.

The class of an Integer literal value is INTEGER.

```
Rational ::= (`-`? [0-9]+ `:` [0-9]+)
```

A Rational literal represents a signed fixed point value with a 16-bit signed integer part and a 16-bit fractional part separated by a colon.

The class of a Rational literal value is RATIONAL.

```
Real ::= (`-`? [0-9]+ `.` [0-9]+ (`e` `-'? [0-9]+)?)
```

A Real literal represents a 32-bit floating point value.

The class of a Real literal value is REAL.

```
Void ::= `void`
```

There is one Void literal values: void.

The class of a Void literal value is NONE.

8.5 Vector literals

```
Vector-Literal ::= {Booleans} | {Characters} | {Integer} | {Rationals} |
{Reals} | {Vector}
```

```
Booleans ::= `{` ( {Boolean} `,`)* {Boolean} `}`
Characters ::= `{` ( {Character} `,`)* {Character} `}`
Integers ::= `{` ( {Integer} `,`)* {Integer} `}`
Rationals ::= `{` ( {Rational} `,`)* {Rational} `}`
Reals ::= `{` ( {Real} `,`)* {Real} `}`
Vector ::= `{` `}`
```

Vector literals represent lists of their corresponding simple literals. They are sequence of simple literals separated by commas and enclosed into braces. The Vector literal is an empty list of Void literal.

The class of a Booleans literal value is BOOLEANS.

The class of a Characters literal value is CHARACTERS.

The class of a Integers literal value is INTEGERS.

The class of a Rationals literal value is RATIONALS.

The class of a Reals literal value is REALS.

The class of a Vector literal value is VECTOR.

It is an error to mix different simple literals in a vector literal.

8.6 String

```
String ::= `"` ([^`"] | `\"` | `\"` | ( `\$` [0-9a-f][0-9a-f]))* `"`
```

The String literal is a lexical construct, no spaces, tabs or returns are allowed in its production.

A String literal represent a sequence of character between double quotes. The double quote and the backslash have to be escaped with a backslash.

Only ASCII characters can be represented directly, all other characters have to be represented by a sequence of backslash, dollar and hexadecimal digits. The sequence of numbers is the UTF-8 code of the character.

The class of a String literal value is STRING.

LIBKEY

1 Introduction

This chapter comments the classes and objects of Libkey, the runtime library of the Key language.

2 Base Classes

2.1 class DUMMY

```
-- self = self is always true  
HAS  
Evaluate()  
  
Ready()  
  
binary = (it)  
  
binary <> (it)
```

2.2 class ANY

```
-- self.Equal(self.Clone()) is always true  
IS  
DUMMY;  
HAS  
Become(theType, ...)  
    theType is? TYPE;  
Clone()  
Conform(theType)  
    theType is? TYPE;  
Default()  
Dub()  
DubFields()  
Echo(theGenerator)  
    theGenerator is? GENERATOR;  
EchoFields(theGenerator)  
    theGenerator is? GENERATOR;
```

```

EchoID(theGenerator)
  theGenerator is? GENERATOR;

EchoIDFields(theGenerator)
  theGenerator is? GENERATOR;

EchoObject(theGenerator)
  theGenerator is? GENERATOR;

Equal(theAny)

Initialize()

NotEqual(theAny)

Supply(theMessage)
  theMessage is? MESSAGE;

Terminate()

```

3 Simple Classes

3.1 class NONE

```

-- void
-- the result of new NONE is void
-- cannot be inherited
-- assigned by value

```

IS

```

ANY;

```

HAS

```

Compare(it)
  it is? NONE;

Dub()

Echo(theGenerator)
  theGenerator is? GENERATOR;

ToNone()

ToString()

```

3.2 class BOOLEAN

```

-- false or true
-- the result of new BOOLEAN is false

```

LIBKEY

```
-- cannot be inherited
-- assigned by value
```

IS

```
ANY;
```

HAS

```
Compare(it)
Echo(theGenerator)
    theGenerator is? GENERATOR;
ToInteger()
ToString()
unary not ()
binary and (it)
    it is? BOOLEAN;
binary or (it)
    it is? BOOLEAN;
binary xor (it)
    it is? BOOLEAN;
```

3.3 class CHARACTER

```
-- 32 bits unicode characters
-- the result of new CHARACTER is '\$00000000'
-- cannot be inherited
-- assigned by value
```

IS

```
ANY;
```

HAS

```
Compare(it, ...)
-- case sensitive if #argument = 1
-- case sensitive if argument @ 2 = true
-- case insensitive if argument @ 2 = false
Echo(theGenerator)
    theGenerator is? GENERATOR;
IsBackspace()
IsBegin()
IsDown()
IsEscape()
IsHelp()
IsLeft()
```

```

IsLower()
IsPageDown()
IsPageUp()
IsReturn()
IsRight()
IsSpace()
IsTab()
IsUp()
IsUpper()
IsXDigit()
Lower()
Upper()
ToInteger()
ToString()
binary - (it)
    it is? CHARACTER;
binary > (it)
    it is? CHARACTER;
binary <= (it)
    it is? CHARACTER;
binary >= (it)
    it is? CHARACTER;

```

3.4 class INTEGER

```

-- 32 bits signed integer numbers
-- the result of new INTEGER is 0
-- cannot be inherited
-- assigned by value

```

IS

```
ANY;
```

HAS

```

Compare(it)
Echo(theGenerator)
    theGenerator is? GENERATOR;
FromBoolean(it)
ToBoolean()

```

```
ToCharacter()  
ToInteger()  
ToRational()  
    self >= -32768;  
    self <= 32767;  
ToReal()  
ToString()  
unary + ()  
unary - ()  
unary ~ ()  
binary * (it)  
    it is? INTEGER;  
binary / (it)  
    it is? INTEGER;  
    it <> 0;  
binary % (it)  
    it is? INTEGER;  
    it <> 0;  
binary + (it)  
    it is? INTEGER;  
binary - (it)  
    it is? INTEGER;  
binary < (it)  
    it is? INTEGER;  
binary > (it)  
    it is? INTEGER;  
binary <= (it)  
    it is? INTEGER;  
binary >= (it)  
    it is? INTEGER;  
binary & (it)  
    it is? INTEGER;  
binary | (it)  
    it is? INTEGER;  
binary ^ (it)  
    it is? INTEGER;
```

3.5 class RATIONAL

```
-- 32 bits signed fixed point numbers
-- the result of new RATIONAL is 0:0
-- cannot be inherited
-- assigned by value
```

IS

```
ANY;
```

HAS

```
Compare(it)
Echo(theGenerator)
  theGenerator is? GENERATOR;
ToInteger()
ToRational()
ToReal()
ToString()
unary + ()
unary - ()
binary * (it)
  it is? RATIONAL;
binary / (it)
  it is? RATIONAL;
  it <> 0:0;
binary + (it)
  it is? RATIONAL;
binary - (it)
  it is? RATIONAL;
binary < (it)
  it is? RATIONAL;
binary > (it)
  it is? RATIONAL;
binary <= (it)
  it is? RATIONAL;
binary >= (it)
  it is? RATIONAL;
```

3.6 class REAL

```
-- 32 bits signed floating point numbers
-- the result of new REAL is 0.0
```

```
-- cannot be inherited
-- assigned by value
```

IS

```
ANY;
```

HAS

```
Compare(it)
Echo(theGenerator)
  theGenerator is? GENERATOR;
ToInteger()
ToRational()
ToReal()
ToString()
unary + ()
unary - ()
binary * (it)
  it is? REAL;
binary / (it)
  it is? REAL;
  it <> 0.0;
binary + (it)
  it is? REAL;
binary - (it)
  it is? REAL;
binary < (it)
  it is? REAL;
binary > (it)
  it is? REAL;
binary <= (it)
  it is? REAL;
binary >= (it)
  it is? REAL;
```

4 Vector Classes

4.1 class VECTOR

```
-- packed array of void
-- the result of new VECTOR is {}
```

```
-- cannot be inherited
-- assigned by reference
```

IS

```
ANY;
```

HAS

```
Compare(theVector, ...)
-- pass ... to compare items
  theVector is? KERNEL.TypeOf(self);

Dub()

Echo(theGenerator)
  theGenerator is? GENERATOR;

Find(theVector, theIndex, ...)
-- return the index of the first occurrence of theVector,
-- starting at theIndex
-- return 0 if not found
-- pass ... to compare items
  theVector is? KERNEL.TypeOf(self);
  theIndex is? INTEGER;
  theIndex > 0;
  theIndex <= #self;

FindAt(theItem, theIndex, ...)
-- return the index of the first occurrence of theItem,
-- starting at theIndex
-- return 0 if not found
-- pass ... to compare items
  theItem is? KERNEL.TypeOfItem(self);
  theIndex is? INTEGER;
  theIndex > 0;
  theIndex <= #self;

GetAt(theIndex)
  theIndex is? INTEGER;
  theIndex > 0;
  theIndex <= #self;

Hash(it)
  it is? INTEGER;

Map(theMessage, ...)

Merge(theVector)
-- return a new VECTOR
  theVector is? KERNEL.TypeOf(self);

Munger(theFromIndex, theToIndex, theSize)
-- move items inside self
-- useful to implement a dynamic vector
  theFromIndex is? INTEGER;
  theToIndex is? INTEGER;
  theSize is? INTEGER;
```

```

    theFromIndex > 0;
    theFromIndex + theSize <= #self + 1;
    theToIndex > 0;
    theToIndex + theSize <= #self + 1;
    theSize >= 0;

PutAt(theItem, theIndex)
    theItem is? KERNEL.TypeOfItem(self);
    theIndex is? INTEGER;
    theIndex > 0;
    theIndex <= #self;

Replace(theVector, theIndex)
-- replace items of self by items of theVector,
-- starting at theIndex
    theVector is? KERNEL.TypeOf(self);
    theIndex is? INTEGER;
    theIndex > 0;
    theIndex + #theVector <= #self + 1;

Reverse()

Take(theIndex, theSize)
-- return a new VECTOR
    theIndex is? INTEGER;
    theSize is? INTEGER;
    theIndex > 0;
    theSize >= 0;
    theIndex + theSize <= #self + 1;

unary # ()

binary % (it)
    it is? INTEGER;

binary - (it)
    it is? KERNEL.TypeOf(self);

binary + (it)
-- return a new VECTOR
    it is? KERNEL.TypeOf(self);

binary < (it)

binary > (it)

binary <= (it)

binary >= (it)

binary @ (it)
    it is? INTEGER;
    it > 0;
    it <= #self;

```

4.2 class **BOOLEANS**

```
-- packed array of false or true
-- the result of new BOOLEANS is {}
-- cannot be inherited
-- assigned by reference
```

IS

```
VECTOR;
```

4.3 class **CHARACTERS**

```
-- packed array of 32 bits unicode characters
-- the result of new CHARACTERS is {}
-- cannot be inherited
-- assigned by reference
```

IS

```
VECTOR;
```

HAS

```
FromString(it)
ToString()
ToCharacters()
```

4.4 class **INTEGERS**

```
-- packed array of 32 bits signed integer numbers
-- the result of new INTEGERS is {}
-- cannot be inherited
-- assigned by reference
```

IS

```
VECTOR;
```

4.5 class **RATIONALS**

```
-- packed array of 32 bits signed fixed point numbers
-- the result of new RATIONALS is {}
-- cannot be inherited
-- assigned by reference
```

IS

```
VECTOR;
```

4.6 class REALS

```
-- packed array of 32 bits signed floating point numbers
-- the result of new REALS is {}
-- cannot be inherited
-- assigned by reference
```

IS

```
VECTOR;
```

5 String Class

5.1 Overview

The STRING class uses UTF-8, i.e. the UCS Transformation Format 8-bit. For instance "a\$FF\$C3"
 If a string is made only of ASCII characters, its UTF-8 version is equal to its ASCII version. All non-ASCII single characters including common Mac OS Roman or Windows Latin accentuated characters require multiple bytes.

So a string is no array of characters and the STRING class has no method to manipulate characters. If you want to use a STRING instance like an array of characters, convert the STRING instance to a CHARACTERS instance.

EXAMPLE

```
result := theString.Count(); -- the number of bytes
result := theString.ToCharacters().Count(); -- the number of characters
```

The STRING class still have methods to manipulate bytes like Compare, Find or Merge.

5.2 class STRING

```
-- UTF-8
-- the result of new STRING is ""
-- cannot be inherited
-- assigned by reference
```

IS

```
ANY;
```

HAS

```
Compare(theString)
  theString is? STRING;

Count()

Dub()

Echo(theGenerator)
  theGenerator is? GENERATOR;
```

```

Hash(it)
  it is? INTEGER;

Merge(theString)
-- return a new STRING
  theString is? STRING;

Munger(theFromIndex, theToIndex, theSize)
-- move items inside self
-- useful to implement a dynamic string
  theFromIndex is? INTEGER;
  theToIndex is? INTEGER;
  theSize is? INTEGER;
  theFromIndex > 0;
  theFromIndex + theSize <= #self + 1;
  theToIndex > 0;
  theToIndex + theSize <= #self + 1;
  theSize >= 0;

Replace(theString, theIndex)
-- replace items of self by items of theString, starting at theIndex
  theString is? STRING;
  theIndex is? INTEGER;
  theIndex > 0;
  theIndex + #theString <= #self + 1;

Resize(theSize)
-- return a new STRING
-- useful to implement a dynamic string
  theSize is? INTEGER;
  theSize >= 0;

Take(theIndex, theSize)
-- return a new STRING
  theIndex is? INTEGER;
  theSize is? INTEGER;
  theIndex > 0;
  theSize >= 0;
  theIndex + theSize <= #self + 1;

ToBoolean()

ToCharacter()

ToCharacters()

ToInteger()

ToMachine()

ToMessage()

ToNone()

ToRational()

ToReal()

```

```

ToString()
ToType()
unary # ()
binary % (it)
    it is? INTEGER;
binary + (it)
-- return a new STRING
    it is? STRING;
binary - (it)
-- return < 0 if self < it, 0 if self = it, > 0 if self > it
    it is? STRING;
binary < (it)
binary > (it)
binary <= (it)
binary >= (it)

```

6 Arrays and Collections

6.1 class ARRAY

```

-- static array
-- the result of new ARRAY is []
-- cannot be inherited
-- assigned by reference

```

IS

```
ANY;
```

HAS

```

Count()
Dub()

EachDown(theMessage, ...)
-- for each item from #self to 1 do .(theMessage)(...)
-- fail when an item does not have theMessage
    theMessage is? MESSAGE;

EachUp(theMessage, ...)
-- for each item from 1 to #self do .(theMessage)(...)
-- fail when an item does not have theMessage
    theMessage is? MESSAGE;

Echo(theGenerator)
    theGenerator is? GENERATOR;

```

```

EchoID(theGenerator)
  theGenerator is? GENERATOR;

FirstThat(theMessage, ...)
-- return the index of the first item from 1 to #self
-- which .(theMessage)(...) = true
-- return 0 if there is no such item
-- fail when an item does not have theMessage or does not return a BOOLEAN
  theMessage is? MESSAGE;

GetAt(theIndex)
  theIndex is? INTEGER;
  theIndex > 0;
  theIndex <= #self;

LastThat(theMessage, ...)
-- return the index of the first item from #self to 1
-- which .(theMessage)(...) = true
-- return 0 if there is no such item
-- fail when an item does not have theMessage or does not return a BOOLEAN
  theMessage is? MESSAGE;

Map(theMessage, ...)
-- replace each item with the result of .(theMessage)(...)
-- fail when an item does not have theMessage
  theMessage is? MESSAGE;

Merge(theArray)
-- return a new ARRAY
  theArray is? ARRAY;

Munger(theFromIndex, theToIndex, theSize)
-- move items inside self
-- useful to implement a dynamic array
  theFromIndex is? INTEGER;
  theToIndex is? INTEGER;
  theSize is? INTEGER;
  theFromIndex > 0;
  theFromIndex + theSize <= #self + 1;
  theToIndex > 0;
  theToIndex + theSize <= #self + 1;
  theSize >= 0;

NextThat(theIndex, theMessage, ...)
-- return the index of the first item from theIndex + 1 to #self
-- which .(theMessage)(...) = true
-- return 0 if there is no such item
-- fail when an item does not have theMessage or does not return a BOOLEAN
  theIndex is? INTEGER;
  theMessage is? MESSAGE;
  theIndex > 0;
  theIndex <= #self;

PreviousThat(theIndex, theMessage, ...)
-- return the index of the first item from theIndex - 1 to 1 which

```

```

.(theMessage)(...) = true
-- return 0 if there is no such item
-- fail when an item does not have theMessage or does not return a BOOLEAN
to theMessage
  theIndex is? INTEGER;
  theMessage is? MESSAGE;
  theIndex > 0;
  theIndex <= #self;

PutAt(theItem, theIndex)
  theIndex is? INTEGER;
  theIndex > 0;
  theIndex <= #self;

Resize(theSize)
-- return a new ARRAY
-- useful to implement a dynamic array
  theSize is? INTEGER;
  theSize >= 0;

Reverse()

Take(theIndex, theSize)
-- return a new ARRAY
  theIndex is? INTEGER;
  theSize is? INTEGER;
  theIndex > 0;
  theSize >= 0;
  theIndex + theSize <= #self + 1;

ToCollection()

unary # ()

binary + (it)
-- return a new ARRAY
  it is? ARRAY;

binary @ (it)
  it is? INTEGER;
  it > 0;
  it <= #self;

```

6.2 class COLLECTION

IS

ANY;

HAS

Items;

Append(theItem)

Count()

```

DubFields()

EachDown(theMessage, ...)
-- for each item from #self to 1 do .(theMessage)(...)
-- fail when an item does not have theMessage
  theMessage is? MESSAGE;

EachUp(theMessage, ...)
-- for each item from 1 to #self do .(theMessage)(...)
-- fail when an item does not have theMessage
  theMessage is? MESSAGE;

EchoFields(theGenerator)

Empty()

FirstThat(theMessage, ...)
-- return the index of the first item from 1 to #self
-- which .(theMessage)(...) = true
-- return 0 if there is no such item
-- fail when an item does not have theMessage or does not return a BOOLEAN
  theMessage is? MESSAGE;

Fit(theSize)
  theSize is? INTEGER;
  theSize >= 0;

GetAt(theIndex)
  theIndex is? INTEGER;
  theIndex > 0;
  theIndex <= self.Count();

IsEmpty()

InsertAt(theItem, theIndex)
  theIndex is? INTEGER;
  theIndex > 0;
  theIndex <= self.Count() + 1;

LastThat(theMessage, ...)
-- return the index of the first item from #self to 1
-- which .(theMessage)(...) = true
-- return 0 if there is no such item
-- fail when an item does not have theMessage or does not return a BOOLEAN
  theMessage is? MESSAGE;

Map(theMessage, ...)
-- replace each item with the result of .(theMessage)(...)
-- fail when an item does not have theMessage
  theMessage is? MESSAGE;

Member(theItem)

Merge(theCollection)
  theCollection is? COLLECTION;

NextThat(theIndex, theMessage, ...)
-- return the index of the first item from theIndex + 1 to #self which

```

```

.(theMessage)(...) = true
-- return 0 if there is no such item
-- fail when an item does not have theMessage or does not return a BOOLEAN
  theIndex is? INTEGER;
  theIndex > 0;
  theIndex <= #self;
  theMessage is? MESSAGE;

Pop()

PreviousThat(theIndex, theMessage, ...)
-- return the index of the first item from theIndex - 1 to 1
-- which .(theMessage)(...) = true
-- return 0 if there is no such item
-- fail when an item does not have theMessage or does not return a BOOLEAN
to theMessage
  theIndex is? INTEGER;
  theIndex > 0;
  theIndex <= #self;
  theMessage is? MESSAGE;

Prepend(theItem)

Push(theItem)

PutAt(theItem, theIndex)
  theIndex is? INTEGER;
  theIndex > 0;
  theIndex <= self.Count();

RemoveAll(theItem)

RemoveAt(theIndex)
  theIndex is? INTEGER;
  theIndex > 0;
  theIndex <= self.Count();

RemoveFirst(theItem)

RemoveLast(theItem)

Resize(theSize)
-- use Fit instead of Resize to propagate change
  theSize is? INTEGER;
  theSize >= 0;

Reverse()

ToCollection()

WithArray(theItems)
-- initialize self with theItems like new COLLECTION.WithArray([]);
  theItems is? ARRAY;

WithSize(theSize)
-- initialize self with theSize like new COLLECTION.WithSize(0);
  theSize is? INTEGER;
  theSize >= 0;

```

```

unary # ( )
binary + (it)
    it is? COLLECTION;
binary @ (theIndex)
    theIndex is? INTEGER;
    theIndex > 0;
    theIndex <= #self;

```

7 Data Classes

7.1 Overview

Instances of data classes are used by C routines to store and retrieve system handles or pointers. Key never touches the system handles or pointers, it is the responsibility of the C routines to manage them.

The Key in C macros `keyFromHandle` and `keyFromPointer` create instances of the `HANDLE` and `POINTER` classes and store a system handle or pointer into them.

The Key in C macros `keyToHandle` and `keyToPointer` retrieve system handles or pointers from instances of the `HANDLE` and `POINTER` classes.

From the Key point of view, there is no difference between the `HANDLE` and `POINTER` classes but both are provided to make sense of the difference between handles and pointers on some systems.

7.2 class HANDLE

```

-- the result of new HANDLE is a system handle which value is 0
-- cannot be inherited
-- assigned by value

```

IS

```
ANY;
```

HAS

```
Dub( )
```

The `Dub` method of the `HANDLE` class returns void since there is no way for Libkey to know how to duplicate the system handle.

7.3 class POINTER

```

-- the result of new POINTER is a system pointer which value is 0
-- cannot be inherited
-- assigned by value

```

LIBKEY

IS

ANY;

HAS

Dub()

The Dub method of the POINTER class returns void since there is no way for Libkey to know how to duplicate the system pointer.

8 Language Classes

8.1 class TYPE

```
-- classes
-- the result of new TYPE is undefined
-- cannot be inherited
-- assigned by value
```

IS

ANY;

HAS

Conforming(it)

Dub()

Echo(theGenerator)

theGenerator is? GENERATOR;

FromString(it)

FromType(it)

MakeFromMessage(theType)

ToType()

8.2 class MESSAGE

```
-- fields or methods
-- the result of new MESSAGE is undefined
-- cannot be inherited
-- assigned by value
```

IS

ANY;

HAS

Dub()

```

Echo(theGenerator)
  theGenerator is? GENERATOR;

Supplying(it)

ToMessage()

ToString()

```

8.3 class MACHINE

```

-- programs
-- the result of new MACHINE is undefined
-- cannot be inherited
-- assigned by value
-- the heap size is the default heap size for each context
-- the stack size is the default stack size for each context
-- the pool size is the number of available context

```

IS

```
DUMMY;
```

HAS

```

GetHeapSize()

TestPoolSize()
  -- result < 0 if the pool size is too small
  -- result > 0 if the pool size is too big

ToString()

```

8.4 class FUTURE

```

-- program results
-- the result of new FUTURE is undefined
-- cannot be inherited
-- assigned by value

```

IS

```
DUMMY;
```

HAS

```

Evaluate()

Ready()

```

8.5 class GENERATOR

IS

```
DUMMY;
```

HAS

```
File;

Tabs;

Usage;

CloseFile()
    self.File is? POINTER;

EchoArray(it)
    it is? ARRAY;
    self.File is? POINTER;

EchoArrayText(it, theMessage)
    it is? ARRAY;
    theMessage is? MESSAGE;
    self.File is? POINTER;

EchoBoolean(it)
    it is? BOOLEAN;
    self.File is? POINTER;

EchoField(it, theField)
    it has? theField;
    self.File is? POINTER;

EchoFieldValue(it, theField, theValue)
    it has? theField;
    self.File is? POINTER;

EchoFieldValueText(it, theField, theValue, theMessage)
    it has? theField;
    theMessage is? MESSAGE;
    self.File is? POINTER;

EchoIDField(it, theField)
    it has? theField;
    self.File is? POINTER;

EchoIDFieldValue(it, theField, theValue)
    it has? theField;
    self.File is? POINTER;

StartOption(theOption, theFlag)
    theOption is? NONE or theOption is? STRING;
    theFlag is? BOOLEAN;
    self.File is? POINTER;

StopOption(theOption, theFlag)
    theOption is? NONE or theOption is? STRING;
    theFlag is? BOOLEAN;
    self.File is? POINTER;

Tab()

Untab()

UseComment()
```

```
UseReturn()
```

```
UseSpace()
```

9 Utilities

9.1 object DEBUGGER

```
-- apply to current context
```

IS

```
DUMMY
```

HAS

```
Break()
```

```
EnableBreak(enableIt)
```

```
IsBreakEnabled()
```

```
Print(theFormat, ...)
```

```
-- check arguments count and type
```

```
-- %c: CHARACTER formatted like printf
```

```
-- %d, %i, %o, %u, %x, %X: INTEGER or RATIONAL formatted like printf
```

```
-- %f, %e, %E, %g, %G: REAL formatted like printf
```

```
-- %p: HANDLE or POINTER formatted like printf
```

```
-- %s: STRING formatted like printf
```

```
-- %k: key formatted like <class object>
```

```
theFormat is? STRING;
```

9.2 object KERNEL

IS

```
DUMMY
```

HAS

```
Cancel()
```

```
ConfigureKOption(theOption)
```

```
theOption is? STRING; -- release option
```

```
Fetch(theName)
```

```
GetError()
```

```
IsLocked()
```

```
-- the current context can modify objects
```

```
LoadFileBuffer(thePath, theBuffer, theSize)
```

```
thePath is? STRING; -- identifier
```

```

    theBuffer is? POINTER;
    theSize is? INTEGER;

LoadPlugin(thePath)
    thePath is? STRING; -- platform dependant path

Lock()
    -- the current context will modify objects

SetError(theError)
    theError is? INTEGER;

Signal(theError)
    theError is? INTEGER;

TestKOption(theOption)
    theOption is? STRING; -- release option

TypeOf(it)

TypeOfItem(it)

Unlock()
    -- the current context will not modify objects

```

9.3 object MEMORY

```
-- apply to current context
```

IS

```
DUMMY
```

HAS

```

CollectGarbage()

IsGarbageCollectionEnabled()

GetHeapCurrentSize()

GetHeapPeakSize()

GetStackCurrentSize()

GetStackPeakSize()

```

KEY IN C

1 Introduction

This chapter annotates the macros and functions of Key in C, the C application programming interface of the Key language and Libkey.

Key in C completes Key. Key in C allows you to define external routines but there is no Key in C functions or macros to define a Key class, object or program in C.

External routines are the only way for Key software to access system features, so that is the main use of Key in C, and that is why Key in C exists.

External routines are sometimes a way to improve the performance of Key softwares, for instance in case of heqavy

2 Files

2.1 Platform Header File

External routines are usually compiled on multiple systems and have often different implementations on each system.

The platform header file, named `platform.h`, defines the macro `mkLinux`, `mkMacOS`, `mkMacOSX`, `mkSolaris` and `mkWin32` as 0 or 1 depending on the platform.

The platform header file is included by the the Key header file.

2.2 Key Header File

The Key header file, named `key.h`, contains the definition of the types, macros and functions that you will use in C to implement external routines.

Every routine C source file of the module has to include the Key header file.

2.3 Module Header File

The C code can use the identifiers of the constants and globals available to the module but you have to tell the Key tool which identifiers your C code needs in the module header file (`.kh`).

The module header file can only contain line of the form

```
keyDeclareID(K_X);
```

where *X* is an identifier available of a constant or a global. The constant or the global can be defined in the module itself or in the modules imported by the module.

Key is case insensitive but *C* is case sensitive. By convention, in *C*, the identifiers are capitalized and preceded by *K_*.

Every routine *C* source file of the module has to include the module header file to use the identifiers.

The type of an identifier in *C* is a *keyID*, it is 32-bit integer that you will pass to some of the function of the Key in *C* API.

For a module named *foo*, the Key tool will look for the module header file named *foo.kh* besides the module source file named *foo.km*.

EXAMPLE

```
keyDeclareID(K_CFOO);
keyDeclareID(K_FOO);
```

2.4 Module C Source File

The Key tool generates the module *C* source file (*.key.c*) which contains the definition of the identifiers, the declaration of the external routines and one *C* function to dispatch the external routines.

For a module named *foo*, the Key tool will generate the module *C* source file named *foo.key.c* in its *C* directory. The name of the *C* function will be *fooMain* or *keyMain* if the module is an application.

EXAMPLE

```
#include "key.h"

keyDefineID(K_CFOO, 0);
keyDefineID(K_FOO, 1);

keyDeclare(cFooFoo);

mkExport void fooMain(tkID theID, tkMachine* the)
{
    static void (*externals[])(tkMachine*) = {
        cFooFoo,
        0
    };
    (*externals[theID])(the);
}
```

2.5 Routine C Source Files

External routines are defined in a *C* source file that has the same directory and the same name as the Key source file where the external routines are declared.

EXAMPLE

```

-- Foo.k
class cFoo
has
    Foo()
    external "cFooFoo";
end;

/* Foo.c */
#include "key.h"
#include "foo.kh"

keyDefine(cFooFoo) {}

```

2.6 Main C Source File

In Key softwares there is always one module that is the application, i.e. a module built with the `-m` option of the Key tool. The application contains both the main C function and the main Key program.

The main C function has to use the `keyRunMain` macro. `keyRunMain` initializes Libkey, runs the main Key program and terminates Libkey.

```

long keyRunMain(long theChunkSize, long theHeapSize, long theStackSize,
                char* theName, void* theReference,
                void (*theKeyMain)(tkID, tkMachine*),
                int argc, char** argv);

```

<code>theFirstSize</code>	the initial size of the memory allocated to Libkey
<code>theNextSize</code>	the incremental size of the memory allocated to Libkey
<code>theKernelSize</code>	the initial size for the constants
<code>theHeapSize</code>	the initial size for the main thread heap
<code>theStackSize</code>	the initial size for the main thread stack
<code>theName</code>	the name of the module
<code>theReference</code>	a system specific reference to the application
<code>theKeyMain</code>	the generated function that dispatch external routines
<code>argc</code>	the number of arguments
<code>argv</code>	the array of arguments
returns	zero if no errors else the error number

Libkey will try to allocate memory from the system when necessary. You can tune how much is allocated initially and how much is at least allocated incrementally with `theFirstSize` and `theNextSize`. Constants, thread heaps and stacks are allocated within the memory allocated to Libkey:

Libkey uses `theReference` to load the module file or resource. On Windows it is the instance of the application, on Mac OS it is the reference number of the resource fork of the application, on Mac OS X it is the main bundle, everywhere else it is the name of the application itself.

`keyRunMain` transforms the arguments into strings and provide them to the program. Pass 0 and NULL if there is no argument.

EXAMPLE

```

int main(int argc, char* argv[])
{
    #if mkWin32

```

```

    return keyRunMain(2048*1024, 1024*1024, 512*1024, 256*1024, 16*1024,
        "foo", GetModuleHandle(NULL), keyMain, argc, argv);
#elif mkMacOS
    return keyRunMain(2048*1024, 1024*1024, 512*1024, 256*1024, 16*1024,
        "foo", (void*)CurResFile(), keyMain, argc, argv);
#elif mkMacOSX
    return keyRunMain(2048*1024, 1024*1024, 512*1024, 256*1024, 16*1024,
        "foo", CFBundleGetMainBundle(), keyMain, argc, argv);
#else
    return keyRunMain(2048*1024, 1024*1024, 512*1024, 256*1024, 16*1024,
        "foo", argv[0], keyMain, argc, argv);
#endif
}

```

3 Routines and Exceptions

3.1 Routine

To define an external routine, use the `keyDefine` macro or the `keyExternal` and `keyContext` macros.

EXAMPLE

```

keyDefine(cFooFoo) {}
keyExternal cFooFoo(keyContext) {}

```

The parameter of an external routine is a pointer to a Key context. It is an opaque structure that you will use only through the macros of Key in C.

Each thread has its own Key context which contains its heap, its registers, its stack, and so on...

Even if the External construct allows any text, external routines are usually named by concatenating the class and the method identifiers.

3.2 Self, Result and Program

To access the instance being called or recalled, external routines use the `keySelf` macro. If the external routine is the routine of a Program construct, the value of `keySelf` is `keyVoid` like the value of `self` is `void` in Key.

To access and assign the result of the routine, external routines use the `keyResult` macro.

To access and assign the storage of the thread, external routines use the `keyProgram` macro.

KEY

```

result := self;
if result = void then end;
program := void;

```

IN C

```
keyResult = keySelf;
if (keyIsVoid(keyResult)) {}
keyProgram = keyVoid;
```

3.3 Arguments

To access arguments, external routines use the `keyArgument` macro. Arguments are accessed by index, like in the `Etc` construct.

`keyArgument(0)` returns the number of arguments like the `Count` construct.

KEY

```
result := argument @ 1;
result := #argument;
```

IN C

```
keyResult = keyArgument(1);
keyResult = keyArgument(0);
```

3.4 Locals

Like internal routines, external routines can declare locals, but, unlike internal routines, external routines have to undeclare locals. The `keyUse` macro allows you to declare and undeclare locals. The routine will fail if the number of declared locals is different than the number of undeclared locals.

In external routines, locals, like arguments, are accessed and assigned by index.

```
key* keyUse(int theCount);
```

`theCount` the number of locals to allocate or deallocate.

KEY

```
Foo()
  use
    a0;
    a1;
  do
    a0 := a1;
  end;
```

IN C

```
keyExternal cFooFoo(keyContext)
{
  key* a = keyUse(2); /* declare two locals */
  a[0] = a[1];
  keyUse(-2); /* undeclare two locals */
}
```

3.5 Exceptions

Key in C provides macros to handle exceptions in external routines.

The `keyDo`, `keyUndo`, `keyRedo` macros are similar to the `do`, `undo` and `redo` keywords. They allow you to structure your C routine like a Key routine.

KEY

```
Foo()
do
undo
    redo;
end;
```

IN C

```
keyExternal cFooFoo(keyContext)
{
    keyDo {
    }
    keyUndo {
        keyRedo;
    }
}
```

To access and assign the value of the exception, external routines use the macro `keyErrorNumber`.

If there is a C break in the `keyUndo` block, the routine will exit normally. It is sometimes a useful way to handle exceptions

EXAMPLE

```
keyDefine(cFooFoo)
{
    keyDo {
    }
    keyUndo {
        break;
    }
    keyPut(keySelf, K_STATUS, keyfromInteger(keyErrorNumber));
    keyErrorNumber = keyNoErrorNumber;
}
```

Libkey itself can trigger exceptions. The value of such exceptions will be `keyFileErrorNumber` or `keyProgramErrorNumber`.

`keyNoErrorNumber`

No error.

`keyFileErrorNumber`

The parser, compiler or linker failed to load a module.

`keyProgramErrorNumber`

An assertion failed or a construct was misused.

3.6 Errors

In Key software, most exceptions will be triggered by external routines.

External routines are the interface between Key and the system. A lot of system functions can fail and have a way to return an error number to the application. Usually, you will trigger an exception if a system function fail. The value of the exception will be the error number.

```
void keyError(int theParameter);
```

theParameter the value of the exception to trigger.

To avoid redundant code in your external routines, you can use one of the `keyIf*` macros. They are platform dependant.

`keyIfError` is more useful on system like Mac OS where a lot of functions returns zero if they succeed and non-zero if they failed.

```
void keyIfError(int theParameter);
```

theParameter if different of zero, trigger an exception.
The value of the exception will be theParameter.

`keyIfFail` is more useful on system like Windows where a lot of functions returns zero if they failed and have a separate way to find the error number.

```
void keyIfFail(int theParameter);
```

theParameter if equal to zero, trigger an exception.
The value of the exception will be paramErr on Mac OS,
GetLastError() on Windows and errno elsewhere.

Since a common failure is the allocation of memory, there is a macro to throw an exception if a pointer or a handle is NULL.

```
void keyIfNULL(void* theParameter);
```

theParameter if equal to NULL, trigger an exception.
The value of the exception will be memFullErr on Mac OS,
ERROR_NOT_ENOUGH_MEMORY on Windows
and ENOMEM elsewhere.

There is also a macro to trigger an exception if a test is false. It works only with the debug version of Libkey.

```
void keyCheck(int theAssertion);
```

theAssertion the assertion to check

EXAMPLE

```
keyIfError(FSRead(aRefNum, &aSize, aBuffer));
keyIfFail(ReadFile(aFile, aBuffer, aSize, &aSize, NULL));
int s = socket(AF_INET, SOCK_STREAM, 0);
keyIfFail(s != -1);
char* p = malloc(1024);
keyIfNULL(p);
keyCheck(p != NULL); /* only debug */
```

3.7 Debugging

Key in C provides macros to help you to debug your C code when you use the debug version of Libkey. Such macros have no effect when you use the release version of Libkey.

```
void keyBreak();
```

`keyBreak` suspends the execution of the current thread and activate `KeysBug`. If Libkey is not connected to `KeysBug`, `keyBreak` does nothing.

```
void keyPrint(char* theFormat, ...);
void keyVPrint(char* theFormat, va_list theArguments);
```

<code>theFormat</code>	format, see <code>printf</code> in in the standard C library
<code>theArguments</code>	variable-length argument list

`keyPrint` and `keyVPrint` print texts in the Work view of `KeysBug`. If Libkey is not connected to `KeysBug`, `keyPrint` and `keyVPrint` print text in `stderr`.

EXAMPLE

```
keyPrint("-- Error: %ld\n", keyErrorNumber);
```

3.8 C Call

If you want to call a C function and pass the Key context, use the `keyCurrentContext` macro and declare your C function with the `keyContext` macro.

EXAMPLE

```
void cFooFooAux(keyContext, int foo)
{
    keyResult = keyFromInteger(foo);
}

keyExternal cFooFoo(keyContext)
{
    cFooFooAux(keyCurrentContext, 1);
}
```

3.9 C Callback

The user interface of some systems is based on C callbacks, for instance window procedures on Windows or event handlers on Mac OS X. Such C callbacks are not nested into the main C function and thus not nested into the main Key program.

Key in C provides the `keyFrame` and `keyUnframe` macros to set up and clean up the Key context of the main Key program. Between `keyFrame` and `keyUnframe`, you can use all the functions of Key in C and `keySelf` equals `keyVoid`.

EXAMPLE

```
LRESULT CALLBACK FooProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    LRESULT result = 0;
```

```

keyFrame {
    switch (msg) {
        case WM_PAINT:
            keyCall0(keyFetch(K_GFOO), K_PAINT);
            break;
        default:
            result = DefWindowProc(hwnd, msg, wParam, lParam);
            break;
    }
}
keyUnframe {
}
return result;
}

```

4 Constructs

4.1 Overview

Part of the functions and macros of the Key in C API are equivalent to expressions or instructions of the Key language. See the Key Language chapter for details.

4.2 keyCall*

keyCall* is the C equivalent of the Call construct.

```

key keyCall0(key theTarget, keyID theMessageID);
key keyCall1(key theTarget, keyID theMessageID, key the0);
key keyCall2(key theTarget, keyID theMessageID, key the0, key the1);
key keyCall3(key theTarget, keyID theMessageID, key the0, key the1,
    key the2);
key keyCall4(key theTarget, keyID theMessageID, key the0, key the1,
    key the2, key the3);
key keyCall5(key theTarget, keyID theMessageID, key the0, key the1,
    key the2, key the3, key the4);
key keyCall6(key theTarget, keyID theMessageID, key the0, key the1,
    key the2, key the3, key the4, key the5);
key keyCall7(key theTarget, keyID theMessageID, key the0, key the1,
    key the2, key the3, key the4, key the5, key the6);
key keyCall8(key theTarget, keyID theMessageID, key the0, key the1,
    key the2, key the3, key the4, key the5, key the6, key the7);

```

theTarget	the instance being called
theMessageID	the keyID of a MESSAGE constant

the*	a Parameter
returns	the result of the executed Routine

KEY

```
result := self.Foo();
```

IN C

```
keyResult = keyCall0(keySelf, K_FOO);
```

4.3 keyConform

keyConform is the C equivalent of the Conform construct.

```
key keyConform(key theTarget, keyID theTypeID);
```

theTarget	the instance being tested
theTypeID	the keyID of a TYPE constant
returns	keyTrue or keyFalse

KEY

```
if self is? cFoo then end;
```

IN C

```
if (keyIsTrue(keyConform(keySelf, K_CFOO))) {}
```

4.4 keyCreate

keyCreate is the C equivalent of the Create construct.

```
key keyCreate(keyID theTypeID);
```

theTypeID	the keyID of a TYPE constant
returns	an instance of the class matching the TYPE constant

KEY

```
result := new cFoo;
```

IN C

```
keyResult := keyCreate(K_CFOO);
```

4.5 keyFetch

keyFetch is the C equivalent of the Fetch construct.

```
key keyFetch(keyID theID);
```

theID	the keyID of a constant or a global
returns	the instance bound to the constant or the instance contained in the global

KEY

```
result := oFoo;
```

IN C

```
keyResult = keyFetch(K_OFOO);
```

4.6 keyGet

keyGet is the C equivalent of the Get construct.

```
key keyGet(key theTarget, keyID theMessageID);
```

theTarget	the instance being read
theMessageID	the keyID of a MESSAGE constant
returns	the instance contained in the field of theTarget

KEY

```
result := self.Foo;
```

IN C

```
keyResult = keyGet(keySelf, K_FOO);
```

4.7 keyPut

keyPut is the C equivalent of the Put construct.

```
void keyPut(key theTarget, keyID theMessageID, key theKey);
```

theTarget	the instance being written
theMessageID	the keyID of a MESSAGE constant
theKey	the instance to be contained in the field of theTarget

KEY

```
self.Foo := void;
```

IN C

```
keyPut(keySelf, K_FOO, keyVoid);
```

4.8 keyRecall*

keyRecall* is the C equivalent of the Recall construct.

```
key keyRecall0(key theTarget, keyID theTypeID, keyID theMessageID);
```

```
key keyRecall1(key theTarget, keyID theTypeID, keyID theMessageID,  
key the0);
```

```
key keyRecall2(key theTarget, keyID theTypeID, keyID theMessageID,  
key the0, key the1);
```

```
key keyRecall3(key theTarget, keyID theTypeID, keyID theMessageID,  
key the0, key the1, key the2);
```

```
key keyRecall4(key theTarget, keyID theTypeID, keyID theMessageID,  
key the0, key the1, key the2, key the3);
```

```
key keyRecall5(key theTarget, keyID theTypeID, keyID theMessageID,
```

```

    key the0, key the1, key the2, key the3,
    key the4);
key keyRecall6(key theTarget, keyID theTypeID, keyID theMessageID,
    key the0, key the1, key the2, key the3,
    key the4, key the5);
key keyRecall7(key theTarget, keyID theTypeID, keyID theMessageID,
    key the0, key the1, key the2, key the3,
    key the4, key the5, key the6);
key keyRecall8(key theTarget, keyID theTypeID, keyID theMessageID,
    key the0, key the1, key the2, key the3,
    key the4, key the5, key the6, key the7);

```

theTarget	the instance being recalled
theTypeID	the keyID of a TYPE constant
theMessageID	the keyID of a MESSAGE constant
the*	a Parameter
returns	the result of the executed Routine

KEY

```
self.cFoo:Foo();
```

IN C

```
keyRecall0(keySelf, K_CFOO, K_FOO);
```

4.9 keyRun*

keyRun* is the C equivalent of the Run construct.

```

key keyRun0(keyID theMachineID);
key keyRun1(keyID theMachineID, key the0);
key keyRun2(keyID theMachineID, key the0, key the1);
key keyRun3(keyID theMachineID, key the0, key the1, key the2);
key keyRun4(keyID theMachineID, key the0, key the1, key the2, key the3);
key keyRun5(keyID theMachineID, key the0, key the1, key the2, key the3,
    key the4);
key keyRun6(keyID theMachineID, key the0, key the1, key the2, key the3,
    key the4, key the5);
key keyRun7(keyID theMachineID, key the0, key the1, key the2, key the3,
    key the4, key the5, key the6);
key keyRun8(keyID theMachineID, key the0, key the1, key the2, key the3,
    key the4, key the5, key the6, key the7);

```

theMachineID	the keyID of a MACHINE Constant
the*	a parameter
returns	an instance of the FUTURE class

KEY

```
result := pFoo();
```

IN C

```
keyResult = keyRun0(K_PFOO);
```

4.10 keyShare

keyShare is the C equivalent of the Share construct.

```
void keyShare(keyID theID);
```

theID the keyID of a global

KEY

```
share gFoo;
```

IN C

```
keyShare(K_GFOO);
```

4.11 keyStore

keyStore is the C equivalent of the Store construct.

```
void keyStore(keyID theID, key theKey);
```

theID the keyID of a global
theKey the instance to be contained in the global

KEY:

```
gFoo := void;
```

IN C

```
keyStore(K_GFOO, keyVoid);
```

4.12 keySupply

keySupply is the C equivalent of the Supply construct.

```
key keySupply(key theTarget, keyID theMessageID);
```

theTarget the instance being tested
theMessageID the keyID of a MESSAGE constant
returns keyTrue or keyFalse

KEY

```
if self has? Foo then end;
```

IN C

```
if (keyIsTrue(keySupply(keySelf, K_FOO))) {}
```

5 Conversions

5.1 Overview

In Key, a value is the instance of a class. In C, such a value is contained in the key structure.

Key in C provide macros to get and set Key values from and to C values.

5.2 Void

```
key keyVoid;
```

keyVoid is the C equivalent of the Void literal.

```
int keyIsVoid(key theKey);
```

theKey	the instance to test
returns	1 if the instance equals void else 0

EXAMPLE

```
keyResult = keyVoid;
```

```
if (keyIsVoid(keyResult)) {}
```

5.3 Boolean

```
key keyFalse;
```

```
key keyTrue;
```

keyFalse and keyTrue are the C equivalent of the Boolean literals.

```
int keyIsFalse(key theKey);
```

theKey	the instance to test
returns	1 if the instance equals false else 0

```
int keyIsTrue(key theKey);
```

theKey	the instance to test
returns	1 if the instance equals true else 0

```
key keyFromBoolean(long theValue);
```

theValue	0 or 1
returns	the BOOLEAN instance

```
long keyToBoolean(key theKey);
```

theKey	the BOOLEAN instance
returns	0 or 1

The debug version of Libkey triggers an exception if theKey is no BOOLEAN instance.

EXAMPLE

```
keyResult = keyFalse;
if (keyIsTrue(keyResult)) {}
keyResult = keyFromBoolean(1);
long one = keyToBoolean(keyResult);
```

5.4 Character

```
key keyFromCharacter(long theValue);
```

theValue	the UCS-4 code
returns	the CHARACTER instance

```
long keyToCharacter(key theKey);
```

theKey	the CHARACTER instance.
returns	the UCS-4 code

The debug version of Libkey triggers an exception if theKey is no CHARACTER instance.

There are functions to convert characters from and to Unicode, see Strings.

EXAMPLE

```
keyResult = keyFromCharacter(0x00000031);
long one = keyToCharacter(keyResult);
```

5.5 Integer

```
key keyFromInteger(long theValue);
```

theValue	the 32-bit signed integer value
returns	the INTEGER instance

```
long keyToInteger(key theKey);
```

theKey	the INTEGER instance
returns	the 32-bit signed integer value

The debug version of Libkey triggers an exception if theKey is no INTEGER instance.

EXAMPLE

```
keyResult = keyFromInteger(1);
long one = keyToInteger(keyResult);
```

5.6 Rational

```
key keyFromRational(long theValue);
```

<code>theValue</code>	the 32-bit signed fixed point value
returns	the RATIONAL instance

```
long keyToRational(key theKey);
```

<code>theKey</code>	the RATIONAL instance
returns	the 32-bit signed fixed point value

The debug version of Libkey triggers an exception if `theKey` is no RATIONAL instance.

EXAMPLE

```
keyResult = keyFromRational(0x00010000);
long one = keyToRational(keyResult);
```

5.7 Real

```
key keyFromReal(float theValue);
```

<code>theValue</code>	the 32-bit floating point value
returns	the REAL instance

```
float keyToReal(key theKey);
```

<code>theKey</code>	the REAL instance
returns	the 32-bit floating point value

The debug version of Libkey triggers an exception if `theKey` is no REAL instance.

EXAMPLE

```
keyResult = keyFromReal(1.0);
long one = keyToReal(keyResult);
```

5.8 Handle

```
key keyFromHandle(void* theValue);
```

<code>theValue</code>	the C handle
returns	the HANDLE instance

```
void* keyToHandle(key theKey);
```

<code>theKey</code>	the HANDLE instance
returns	the C handle

The debug version of Libkey triggers an exception if `theKey` is no HANDLE instance.

EXAMPLE

```
keyResult = keyFromHandle(NewHandle(1));
Handle one = keyToHandle(keyResult);
```

5.9 Pointer

```
key keyFromPointer(void* theValue);

    theValue          the C pointer
returns             the POINTER instance

void* keyToPointer(key theKey);

    theKey            the POINTER instance
returns             the C pointer
```

The debug version of Libkey triggers an exception if `theKey` is no POINTER instance.

EXAMPLE

```
keyResult = keyFromPointer(NewPtr(1));
Ptr one = keyToPointer(keyResult);
```

5.10 Type

```
key keyFromType(keyID theTypeID);

    theTypeID          the keyID of a TYPE constant
returns             the TYPE instance
```

The debug version of Libkey triggers an exception if `theTypeID` is no ID of a TYPE constant. Otherwise `keyFromType` is equivalent to `keyFetch`.

`keyToType` would be ambiguous since some TYPE constants have no keyID for the module. `keyFromType` is useful to pass a TYPE constant as a parameter.

EXAMPLE

```
keyResult = keyFromType(K_CFOO);
```

5.11 Message

```
key keyFromMessage(keyID theMessageID);

    theMessageID      the keyID of a MESSAGE constant
returns             the MESSAGE instance
```

The debug version of Libkey triggers an exception if `theMessageID` is no ID of a MESSAGE constant. Otherwise `keyFromMessage` is equivalent to `keyFetch`.

`keyToMessage` would be ambiguous since some MESSAGE constants have no keyID for the module. `keyFromMessage` is useful to pass a MESSAGE instance as a parameter.

EXAMPLE

```
keyResult = keyFromMessage(K_FOO);
```

5.12 Machine

```
key keyFromMachine(keyID theMachineID);
```

theMachineID	the keyID of a MACHINE constant
returns	the MACHINE instance

The debug version of Libkey triggers an exception if `theMachineID` is no ID of a MACHINE constant. Otherwise `keyFromMachine` is equivalent to `keyFetch`.

`keyToMachine` would be ambiguous since some MACHINE constants have no keyID for the module. `keyFromMachine` is useful to pass a MACHINE instance as a parameter.

EXAMPLE

```
keyResult = keyFromMachine(K_PFOO);
```

5.13 Vector Address

```
void* keyLockVector(key theKey);
```

theKey	the VECTOR instance
returns	the address to the content

```
void keyUnlockVector(key theKey);
```

theKey	the VECTOR instance
--------	---------------------

The debug version of Libkey triggers an exception if `theKey` is no VECTOR instance.

You can peek and poke the content of a VECTOR instance between `keyLockVector` and `keyUnlockVector`. The garbage collector of the current thread is disabled by `keyLockVector` and enabled by `keyUnlockVector`.

Be sure to take into account the size of the VECTOR instance because nothing prevents you to peek and poke outside of its content.

The content is an array of 32-bit values. Depending on the class of VECTOR, you can cast its address to `long*`, `float*`, and so on...

EXAMPLE

```
/* assume keyResult is an instance of INTEGERS */
long i, c = keyToInteger(keyCall0(keyResult, K_COUNT));
long* p = keyLockVector(keyResult);
for (i = 0; i < c; i++) *p++ = 1;
keyUnlockVector(keyResult);
```

6 Strings

6.1 Overview

A `STRING` instance is sequence of characters encoded in UTF-8. Most systems use a different encoding, usually dependant on its localization.

Key in C provides macros to convert from strings to `STRING` instances and vice versa.

All the macros with `Script`, `System` or `Unicode` in their names transcode strings from and to UTF-8 and can fail. The macros without `Script`, `System` or `Unicode` in their names do not transcode strings. They are useful for ASCII and UTF-8 strings.

What a script is depends on the system. On Mac OS and Mac OS X, it is a Script Manager enum, for instance `smJapanese`. On Windows, it is a code page, for instance 932.

Using the macros with `System` in their name is equivalent to using the macros with `Script` in their name and passing the script the system is currently running.

6.2 From C To Key

```
key keyFromCString(char* theString)
key keyFromPascalString(unsigned char* theString);
key keyFromString(unsigned char* theString, size_t theSize);
key keyFromScriptCString(long theScript, char* theString);
key keyFromScriptPascalString(long theScript, unsigned char* theString);
key keyFromScriptString(long theScript, unsigned char* theString,
    size_t theSize);
key keyFromSystemCString(char* theString);
key keyFromSystemPascalString(unsigned char* theString);
key keyFromSystemString(unsigned char* theString, size_t theSize);
key keyFromUnicodeString(unsigned short* theString, size_t theSize);
```

<code>theScript</code>	a system specific value
<code>theString</code>	the address of the string
<code>theSize</code>	the size of the string
returns	the <code>STRING</code> instance

```
key keyFromScriptCharacter(long theScript, unsigned char* theString,
    size_t theSize);
key keyFromSystemCharacter(unsigned char* theString, size_t theSize);
```

<code>theScript</code>	a system specific value
<code>theString</code>	the address of the character
<code>theSize</code>	the size of the character
returns	the <code>CHARACTER</code> instance

`keyToScriptCharacter` and `keyToSystemCharacter` have a programming interface like strings because scripts can use several bytes to represent a character.

6.3 From Key To C

```

size_t keyToCString(key theKey, char* theString, size_t theSize);
size_t keyToPascalString(key theKey, unsigned char* theString,
    size_t theSize);
size_t keyToString(key theKey, unsigned char* theString, size_t theSize);
size_t keyToScriptCString(key theKey, long theScript,
    char* theString, size_t theSize);
size_t keyToScriptPascalString(key theKey, long theScript,
    unsigned char* theString, size_t theSize);
size_t keyToScriptString(key theKey, long theScript,
    unsigned char* theString, size_t theSize);
size_t keyToSystemCString(key theKey, char* theString, size_t theSize);
size_t keyToSystemPascalString(key theKey, unsigned char* theString,
    size_t theSize);
size_t keyToSystemString(key theKey, unsigned char* theString,
    size_t theSize);
size_t keyToUnicodeString(key theKey, unsigned short* theString,
    size_t theSize);

```

theKey	the STRING instance
theScript	a system specific value
theString	the buffer where the string will be converted
theSize	the size of the buffer
returns	the size of the string

The debug version of Libkey triggers an exception if theKey is no STRING instance.

theSize and the returned size include the zero byte at end of C strings and the length byte at the beginning of Pascal strings.

```

size_t keyToScriptCharacter(key theKey, long theScript,
    unsigned char* theString, size_t theSize);
size_t keyToSystemCharacter(key theKey, unsigned char* theString,
    size_t theSize);

```

theKey	the CHARACTER instance
theScript	a system specific value
theString	the buffer where the character will be converted
theSize	the size of the buffer
returns	the size of the character

The debug version of Libkey triggers an exception if theKey is no CHARACTER instance.

keyToScriptCharacter and keyToSystemCharacter have a programming interface like strings because scripts can use several bytes to represent a character.

6.4 From Key To C size

```

size_t keyToStringSize(key theKey);
size_t keyToScriptStringSize(key theKey, long theScript);

```

```
size_t keyToSystemStringSize(key theKey);
size_t keyToUnicodeStringSize(key theKey);
```

theKey	the STRING instance
theScript	a system specific value
returns	the size of the buffer that the matching keyTo*String macro needs.

The debug version of Libkey triggers an exception if `theKey` is no STRING instance.

The returned size includes the zero byte at end of C strings and the length byte at the beginning of Pascal strings.

6.5 String Address

```
void* keyLockString(key theKey);
```

theKey	the STRING instance
returns	the address to the content

```
keyUnlockString(key theKey);
```

theKey	the STRING instance
--------	---------------------

The debug version of Libkey triggers an exception if `theKey` is no STRING instance.

You can peek and poke the content of a STRING instance between `keyLockString` and `keyUnlockString`. The garbage collector of the current thread is disabled by `keyLockString` and enabled by `keyUnlockString`.

Be sure to take into account the size of the STRING instance because nothing prevents you to peek and poke outside of its content.

The content is an array of 8-bit values, the UTF-8 codes of the STRING instance.

EXAMPLE

```
/* assume keyResult is an instance of STRING */
long i, c = keyToInteger(keyCall0(keyResult, K_COUNT));
char* p = keyLockString(keyResult);
for (i = 0; i < c; i++) *p++ = '1';
keyUnlockString(keyResult);
```

7 Resources

```
void* keyOpenResources();
```

```
void keyCloseResources(void* theResources);
```

KEYSBUG

1 Introduction

This chapter describes Keysbug, the interactive debugger of the Key language. Libkey and Keysbug communicates through TCP-IP so you can debug Key softwares locally or remotely.

2 To be continued...