XCLE

Tutorials V 1.2.5

eXtensible Concatenative Language Engine

XCLE: Programmatic handling of executable code, combining easy generation and manipulation with execution speed and memory efficiency at runtime.

Tutorials

XCLE - Basic structure of XCLE objects and programs.

Introduction

XCLE was originally started as OKit, a basic object manipulation toolkit for use in Genetic Programming projects, that soon evolved toward a full-fledged language and compiler.

While most of the code and object-handling algorithms have been changed with respect to the OKit libraries, the goal remains essentially the same: to provide a two-level executable code manipulation system, by building the code programmatically, or by compiling human-readable and -writable sources.

Anything that XCLE can manipulate is an object, with a complete set of methods to create, edit and delete internal data. And most of these can also be specified inline, in a character stream, and parsed to produce binary object. This second method relies on "Primitives Libraries", or definitions of a set of low-level instructions, called thereon "primitives". These primitives associate a name with informations related to execution (think "C-prototype") and machine-code that realize the actual operation.

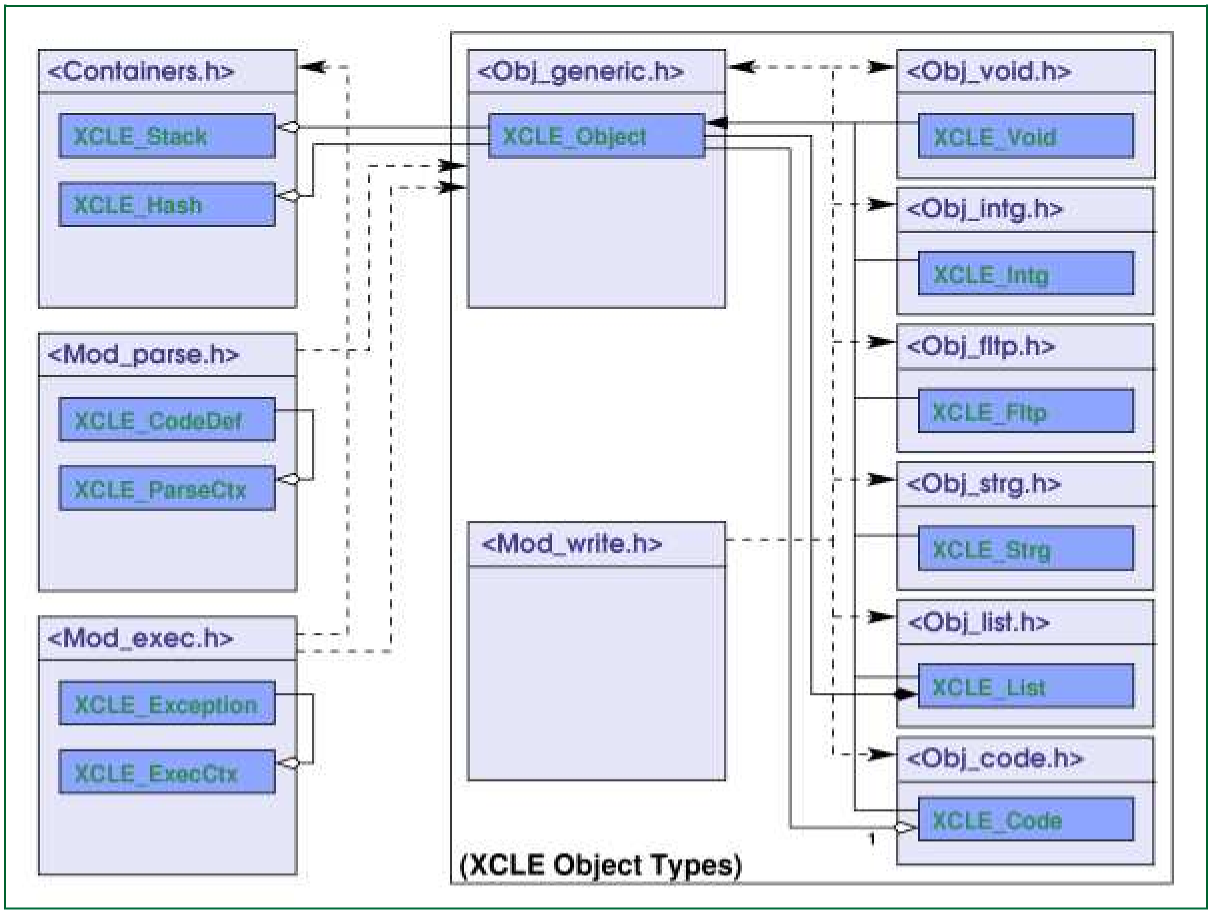
We will attempt in this tutorial to show how these objects interact, how they are produced and used, from a language-user (as opposed to developer) perspective.

Syntax of XCLE programs

Types

XCLE knows how to handle a few predefined types: integers, floats, strings, lists, and primitives. These are objects, sharing a common "generic" ancestor. Integers are 32bit signed integers, floats are double precision floating-point numbers, lists are arbitrarily long vectors of generic objects, and primitives are the base of the ’executable’ feature of these programs. All these objects are memory-managed, which means that, except when programming new primitives, very little is to be done to ensure proper memory allocation and waste-collection.

XCLE Class Diagram



Most, if not all, data types can be implemented in terms of combinations of these structures. And objects imported from C can be integrated in XCLE using a numeric pointer value as an integer object in XCLE. If having a dedicated type for a given data structure is an issue (e.g. to ensure that the data type is accessed only by the proper handlers), this can be obtained using a dedicated primitive, in a procedure more extensively detailed in a further document (see the Extending XCLE tutorial).

Syntax

Syntaxically, XCLE programs are blank-separated successions of strings, lists, primitives and barewords (i.e. blocks of alphanumeric characters without delimiters).

Integers are all-numeric barewords (containing only the characters 0123456789+-).

Floats are barewords containing numeric characters, or the characters '.', the floationg point marker, and 'e' or 'E', the exponent marker.

Strings are double-quote delimited successions of characters, allowing for escaped special characters (\a,\f,\r,\n,\t,\") and octal escapes (of the form \xxx).

Lists are simply an XCLE-parsable succession of characters, leading to a valid list of objects, encased in square brackets ('[' and ']').

Primitives take the form '<name>', or '<name:data>', where 'name' is a group of characters without whitespace nor XCLE delimiters (i.e. <>[]:"), and 'data' is a --single-- valid object. In this form, they need not be defined, that is, have defined prototype and machine-code properties.

Unaccounted-for barewords (those that are not integers or floats) are first parsed as primitives (like <bareword>, but only if this exists as a defined primitive), and failing that as strings.

An example

Here is an instance of a syntaxically valid XCLE program, in its string representation:

```
[ "three: " .4e+1 -1 <+> <dupN:2> <tostr> <strcat> ]
```

Executing XCLE programs

Execution Model

XCLE makes no distinction between executable code and data. Programs are lists, containing objects of the predefined types. Upon execution of such a list/program, each object in turn is deposited on a stack, except for primitives.

Primitives are checked against the current state of the stack (as defined by the objects presents before execution, and those deposited during it). If the test fails, an exception is raised. Otherwise, the machine-code section of the primitive gets executed, and the remaining part of the list is examined.

This step-by-step processing, reminding of the RPL syntax (operators are cited after their arguments), will be called hereafter the "execution stream". At each step, the contents of the stack are critical to the execution of the remains of the stream, both in the necessary sense (i.e. the arguments needed to successfully complete the execution) and the sufficent sense (i.e., the end of the execution stream behaves only according to the arguments supplied, whatever the means -- the first part of the program, or objets put on the stack by hand).

A concrete example

Assuming primitives called '+', '**dupN**', '**tostr**' and '**strcat**' have been defined, and behave like their name would lead us to expect (respectiveley, numeric addition, replicating a given number of arguments on the stack, stringification and string concatenation) the above program would thus be executed in the following way:

- the string **"three: "** is pushed on the stack
- the float **4.0e+00** is pushed on the stack
- the integer **-1** is pushed on the stack
- the '+' primitive is executed, taking **4.0e+00** and **-1** from the stack and leaving the float **3.0e+00**
- the '**dupN**' primitive is executed, with parameter **2**, replicating the two first levels of the stack
- the '**tostr**' primitive is executed, the first stack level **3.0e+00** becoming the string **"3.0e+00"**
- the '**strcat**' primitive is executed, making **"three: 3.0e+00"** out of **"three: "** and **"3.0e+00"**
- the execution terminates successfully

Parsing and execution contexts

Primitives definitions

XCLE by itself defines no primitive. The syntax allows you to define named primitives using the <name> construct, created, by default, empty. Only already defined primitives will have actual content, meaning their execution will result in anything else than keeping the stack unchanged.

As we see, defining primitives is thus necessary for XCLE programs to do anything useful. This can be done in two ways. The first gives you complete control on the primitives definitions, using API calls to build a primitive template, and register it with the parsing tools. This process is described in depth in the second tutorial, Using the XCLE API.

However, the easiest method is to load a predefined module (that is, a file containing binary definitions of several primitives). This can be done via API calls, or by using command-line options with the XCLE compiler cxcl,

A standard primitives module for XCLE, XCLstd, defines a wide set of basic operations on predefined objects, like stack operations, string and list toolkits, standard mathematic functions.

If a particular set of operations is needed, or the standard set is too large, custom modules can be defined. This process is explained in the Extending XCLE part of this tutorial.

Practical use of modules

Let us assume we have defined the '+', '**dupN**', '**tostr**' and '**strcat**' primitives we needed for our above examples, in a module named "MyTest". The effective file name will more probably be "MyTest.so" (or "MyTest.dll" on Windows), since the modules are actually shared libraries. We will assume this file is in the current working directory, or in the library search path. We will use cxcl to execute our program, through the command line:

```
cxcl -L -l "MyTest.so" '[ "three: " .4e+1 -1 <+> <dupN:2> <tostr> <strcat> ]'
```

The -L switch tells cxcl not to load the default module, XCLstd. We then specify the module we want loaded, with the -l switch. The first non-switch argument is our program, that will be parsed using the module's definitions, then executed.

The output shows the execution status of the program and the resulting stack contents:

```
Evaluated [ "three: " 4.0e+00 -1 <+> <dupN:2> <tostr> <strcat> ] ; OK

.= STACK ==================================================================.
| 00008 :                                                   |
| 00007 :                                                   |
| 00006 :                                                   |
| 00005 :                                                   |
| 00004 :                                                   |
| 00003 :                                        "three: " |
| 00002 :                                          3.0e+00 |
| 00001 :                                   "three: 3.0e+00" |
*==========================================================================*
```

XCLE - Using the XCLE API to manipulate and evaluate structures.

Structure of the API

Use of DbgLog

DbgLog filters and logs debugging/error messages according to errno, calling module/function, warning keyword and debugging level. Particular errno/keyword combinations can be made fatal. Output log files depend on keyword.

Error tracking

To use **DbgLog**, simply **#include** <DbgLog.h>.

Error reporting can be done using:
void DbgLog(const char * mod, const char * key, unsigned char lev, const char * mes, ...) ;

’mod’ is the calling module or function name. This is the main context data, as this information will structure the actual hierarchy of calls. ’key’ is a debugging key, an identifier tag that will direct logging and filtering of this message. ’lev’ is a numeric indication of severity (the lower, the more severe, with 0 having sense as "system-level error").

For instance:
DbgLog("MyFunction","dummyerror",16,"This is my error number %d",num++) ;

Reporting levels

DbgLog_UpDbgLvl and **DbgLog_DnDbgLvl** respectively toggle on and off displaying messages from the error levels given by the set bits of the mask (1 being the level 0, 1<<1 the level 1, and then on until 1<<31).

Error stack and recursive calls

DbgLog_OpenBlock and **DbgLog_CloseBlock** are used to open and close trapping blocks, that is to suspend error reporting. No messages will be printed, except when **DbgLog_Flush** is called, that will print all trapped error entries from the last call to **DbgLog_OpenBlock**.

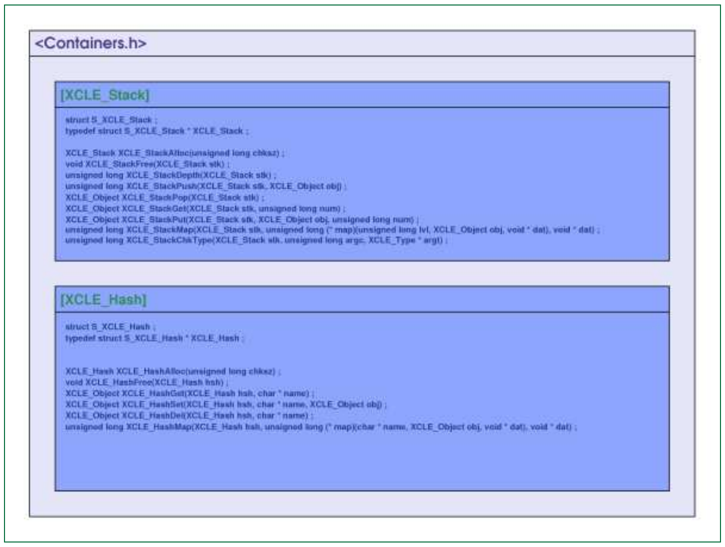
Managing imbricated failures

When using imbricated calls to functions using DbgLog, the uppermost can fail because one lower call failed. In this case, using DbgLog again would result in two messages for the same material error.. While this may be desirable, most often this would only be confusing at debug time. It is advisable to limit the use of DbgLog on generation of error, that is: a failed system call (so the errno variable is set to a meaningfull value), a check that showed some discrepancy in data, wrong arguments...

Support features

Two complex structures are used to store objects during execution. These are the **XCLE_Stack** and **XCLE_Hash** objects, defined in **<XCLE/Containers.h>**.

XCLE/Containers.h definitions



```
struct S_XCLE_Stack ;
typedef struct S_XCLE_Stack * XCLE_Stack ;

XCLE_Stack XCLE_StackAlloc(unsigned long chksz) ;
void XCLE_StackFree(XCLE_Stack stk) ;
unsigned long XCLE_StackDepth(XCLE_Stack stk) ;
unsigned long XCLE_StackPush(XCLE_Stack stk, XCLE_Object obj) ;
XCLE_Object XCLE_StackPop(XCLE_Stack stk) ;
XCLE_Object XCLE_StackGet(XCLE_Stack stk, unsigned long num) ;
XCLE_Object XCLE_StackPut(XCLE_Stack stk, XCLE_Object obj, unsigned long num) ;
unsigned long XCLE_StackMap(XCLE_Stack stk, unsigned long (* map)(unsigned long lvl, XCLE_Object obj, void * dat), void * dat) ;
unsigned long XCLE_StackChkType(XCLE_Stack stk, unsigned long argc, XCLE_Type * argt) ;
```

```
struct S_XCLE_Hash ;
typedef struct S_XCLE_Hash * XCLE_Hash ;


XCLE_Hash XCLE_HashAlloc(unsigned long chksz) ;
void XCLE_HashFree(XCLE_Hash hsh) ;
XCLE_Object XCLE_HashGet(XCLE_Hash hsh, char * name) ;
XCLE_Object XCLE_HashSet(XCLE_Hash hsh, char * name, XCLE_Object obj) ;
XCLE_Object XCLE_HashDel(XCLE_Hash hsh, char * name) ;
unsigned long XCLE_HashMap(XCLE_Hash hsh, unsigned long (* map)(char * name, XCLE_Object obj, void * dat), void * dat) ;
```

Stack

The **XCLE_Stack** object holds arguments and results while executing programs. It can be manipulated by poping and pushing data (with **XCLE_StackPop** and **XCLE_StackPush** respectiveley) on the lowermost stack level, thus lowering/raising other objects already on the stack. An object on a given level can be accessed through **XCLE_StackGet**, that returns the object without removing it from the stack.

Hash

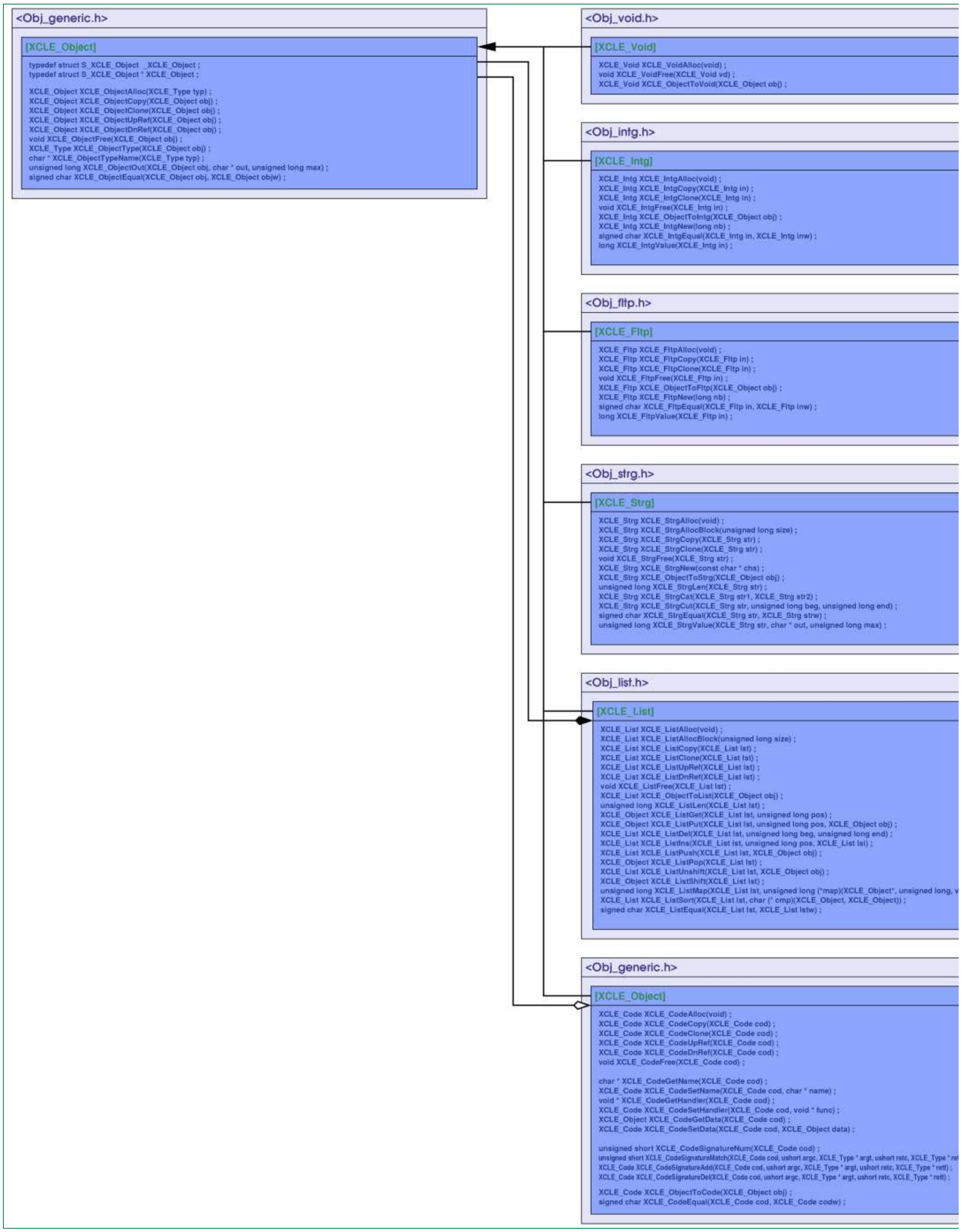
The **XCLE_Hash** object can be used as a variables table. It holds (name,object) pairs, giving access to the objects by their name in an efficient way. **XCLE_HashSet**, **XCLE_HashGet** and **XCLE_HashDel** provide ways to manipulate this association, by respectively creating/modifing an entry, retriving the contents of an entry, or deleting an entry.

Creating objects

XCLE defines several language objects types, that is, object that will be accessible as types inside the XCLE programming language. These are **XCLE_Void**, used as an undefined value, **XCLE_Intg** and **XCLE_Fltp**, providing the integer and floating point numeric types, **XCLE_Strg** that holds character string or binary data, **XCLE_List** which serves both as a collection structure, and a program structure, and **XCLE_Code** that provides the language's basic instructions, or primitives. These types inherit from a generic object type **XCLE_Object**, which stands for any language-level object.

No data is directly accessible inside these objects. Instead, constructors and accessors are provided, that check data integrity and help making memory management transparent. Object creation can happen in two ways: the use of an explicit constructor, of the form XCLE_<type>New(data), or the parsing of a valid string representation, through the **XCLE_ObjectParse** and **XCLE_ListParse** methods defined in the **<Mod_parse.h>** header file.

Language types definitions



```
typedef struct S_XCLE_Object _XCLE_Object ;
typedef struct S_XCLE_Object * XCLE_Object ;

XCLE_Object XCLE_ObjectAlloc(XCLE_Type typ) ;
XCLE_Object XCLE_ObjectCopy(XCLE_Object obj) ;
XCLE_Object XCLE_ObjectClone(XCLE_Object obj) ;
XCLE_Object XCLE_ObjectUpRef(XCLE_Object obj) ;
XCLE_Object XCLE_ObjectDnRef(XCLE_Object obj) ;
void XCLE_ObjectFree(XCLE_Object obj) ;
XCLE_Type XCLE_ObjectType(XCLE_Object obj) ;
char * XCLE_ObjectTypeName(XCLE_Type typ) ;
unsigned long XCLE_ObjectOut(XCLE_Object obj, char * out, unsigned long max) ;
signed char XCLE_ObjectEqual(XCLE_Object obj, XCLE_Object objw) ;
```

```
XCLE_Void XCLE_VoidAlloc(void) ;
void XCLE_VoidFree(XCLE_Void vd) ;
XCLE_Void XCLE_ObjectToVoid(XCLE_Object obj) ;
```

```
XCLE_Intg XCLE_IntgAlloc(void) ;
XCLE_Intg XCLE_IntgCopy(XCLE_Intg in) ;
XCLE_Intg XCLE_IntgClone(XCLE_Intg in) ;
void XCLE_IntgFree(XCLE_Intg in) ;
XCLE_Intg XCLE_ObjectToIntg(XCLE_Object obj) ;
XCLE_Intg XCLE_IntgNew(long nb) ;
signed char XCLE_IntgEqual(XCLE_Intg in, XCLE_Intg inw) ;
long XCLE_IntgValue(XCLE_Intg in) ;
```

```
XCLE_Fltp XCLE_FltpAlloc(void) ;
XCLE_Fltp XCLE_FltpCopy(XCLE_Fltp in) ;
XCLE_Fltp XCLE_FltpClone(XCLE_Fltp in) ;
void XCLE_FltpFree(XCLE_Fltp in) ;
XCLE_Fltp XCLE_ObjectToFltp(XCLE_Object obj) ;
XCLE_Fltp XCLE_FltpNew(long nb) ;
signed char XCLE_FltpEqual(XCLE_Fltp in, XCLE_Fltp inw) ;
long XCLE_FltpValue(XCLE_Fltp in) ;
```

```
XCLE_Strg XCLE_StrgAlloc(void) ;
XCLE_Strg XCLE_StrgAllocBlock(unsigned long size) ;
XCLE_Strg XCLE_StrgCopy(XCLE_Strg str) ;
XCLE_Strg XCLE_StrgClone(XCLE_Strg str) ;
void XCLE_StrgFree(XCLE_Strg str) ;
XCLE_Strg XCLE_StrgNew(const char * chs) ;
XCLE_Strg XCLE_ObjectToStrg(XCLE_Object obj) ;
unsigned long XCLE_StrgLen(XCLE_Strg str) ;
XCLE_Strg XCLE_StrgCat(XCLE_Strg str1, XCLE_Strg str2) ;
XCLE_Strg XCLE_StrgCut(XCLE_Strg str, unsigned long beg, unsigned long end) ;
signed char XCLE_StrgEqual(XCLE_Strg str, XCLE_Strg strw) ;
unsigned long XCLE_StrgValue(XCLE_Strg str, char * out, unsigned long max) ;
```

```
XCLE_List XCLE_ListAlloc(void) ;
XCLE_List XCLE_ListAllocBlock(unsigned long size) ;
XCLE_List XCLE_ListCopy(XCLE_List lst) ;
XCLE_List XCLE_ListClone(XCLE_List lst) ;
XCLE_List XCLE_ListUpRef(XCLE_List lst) ;
XCLE_List XCLE_ListDnRef(XCLE_List lst) ;
void XCLE_ListFree(XCLE_List lst) ;
XCLE_List XCLE_ObjectToList(XCLE_Object obj) ;
unsigned long XCLE_ListLen(XCLE_List lst) ;
XCLE_Object XCLE_ListGet(XCLE_List lst, unsigned long pos) ;
XCLE_Object XCLE_ListPut(XCLE_List lst, unsigned long pos, XCLE_Object obj) ;
XCLE_List XCLE_ListDel(XCLE_List lst, unsigned long beg, unsigned long end) ;
XCLE_List XCLE_ListIns(XCLE_List lst, unsigned long pos, XCLE_List lsl) ;
XCLE_List XCLE_ListPush(XCLE_List lst, XCLE_Object obj) ;
XCLE_Object XCLE_ListPop(XCLE_List lst) ;
XCLE_List XCLE_ListUnshift(XCLE_List lst, XCLE_Object obj) ;
XCLE_Object XCLE_ListShift(XCLE_List lst) ;
unsigned long XCLE_ListMap(XCLE_List lst, unsigned long (*map)(XCLE_Object*, unsigned long, v
XCLE_List XCLE_ListSort(XCLE_List lst, char (* cmp)(XCLE_Object, XCLE_Object)) ;
signed char XCLE_ListEqual(XCLE_List lst, XCLE_List lstw) ;
```

```
XCLE_Code XCLE_CodeAlloc(void) ;
XCLE_Code XCLE_CodeCopy(XCLE_Code cod) ;
XCLE_Code XCLE_CodeClone(XCLE_Code cod) ;
XCLE_Code XCLE_CodeUpRef(XCLE_Code cod) ;
XCLE_Code XCLE_CodeDnRef(XCLE_Code cod) ;
void XCLE_CodeFree(XCLE_Code cod) ;

char * XCLE_CodeGetName(XCLE_Code cod) ;
XCLE_Code XCLE_CodeSetName(XCLE_Code cod, char * name) ;
void * XCLE_CodeGetHandler(XCLE_Code cod) ;
XCLE_Code XCLE_CodeSetHandler(XCLE_Code cod, void * func) ;
XCLE_Object XCLE_CodeGetData(XCLE_Code cod) ;
XCLE_Code XCLE_CodeSetData(XCLE_Code cod, XCLE_Object data) ;

unsigned short XCLE_CodeSignatureNum(XCLE_Code cod) ;
unsigned short XCLE_CodeSignatureMatch(XCLE_Code cod, ushort argc, XCLE_Type * argt, ushort retc, XCLE_Type * ret
XCLE_Code XCLE_CodeSignatureAdd(XCLE_Code cod, ushort argc, XCLE_Type * argt, ushort retc, XCLE_Type * rett) ;
XCLE_Code XCLE_CodeSignatureDel(XCLE_Code cod, ushort argc, XCLE_Type * argt, ushort retc, XCLE_Type * rett) ;

XCLE_Code XCLE_ObjectToCode(XCLE_Object obj) ;
signed char XCLE_CodeEqual(XCLE_Code cod, XCLE_Code codw) ;
```

The XCLE_<type>New(data) methods

For a few objects, the XCLE_<type>New(data) methods do not exist, and are replaced by XCLE_<type>Alloc() methods that take no argument.

The **XCLE_VoidAlloc** method creates a new **XCLE_Void** object. It takes no argument.

The **XCLE_IntgNew** method creates a new **XCLE_Intg** object. It takes a single argument, the integer number whose value will take the newly created object.

The **XCLE_FltpNew** method creates a new **XCLE_Fltp** object. It takes a single argument, the floating point number that will serve to initialize the object.

The **XCLE_StrgNew** method creates a new **XCLE_Strg** object. It takes a single argument, a C-style character string (i.e. a nul-char-terminated character array) copied into the new object.

The **XCLE_ListAlloc** method creates a new **XCLE_List** object. It takes no argument, and returns an empty list.

The **XCLE_CodeAlloc** should probably be used only by primitives library developers. It is useless per se, as the returned **XCLE_Code** object is devoid of any information such as primitive name, or handler. Executing a program containing this object will result in a runtime error. **XCLE_Code** objects should be created using the appropriate methods from the **<XCLE/Mod_Parse.h>** module, as detailed in the Parsing process section.

String parsing

Provided a parsing context has been defined, containing primitives definitions and parsing parameters, the most simple way of creating objects is probably to parse them from their string representation, using methods such as **XCLE_ObjectParse** and **XCLE_ListParse**, whose use will be detailed in the Parsing section.

In the same context, the **XCLE_ParseCtx_CodeByName** method can be used instead of **XCLE_CodeAlloc** to obtain valid **XCLE_Code** objects. It asks for a defined parsing context, and a string that should be the name of a primitive registered in the parsing context. A fully functionnal **XCLE_CodeAlloc** object will be returned.

Reference counting and memory

The memory management system of XCLE reposes on references count. Each object has its own, that reflect the number of "ownership tokens" that were distributed for this object. Each module or code structure that makes use of an object and is susceptible to share it with other structures must have one such token on the object.

Increasing the reference count of an object means acquiring such a token, while relinquising this token bars this code section from any right to use this object anymore.

An object can be definitively freed only if its reference count is zero, that is, if no part of the program is using this object anymore.

Be wary of implicit (de-)referencing of objects included in compounds (lists elements, and code data segment). It will happen when the references on parent objects are modified, because ownership of the parent means also ownership of its components.

Creation

As we saw above, most types have their **XCLE_<Type>Alloc** constructors, or alternately **XCLE_<Type>New**. However, calling these constructors isn't enough to reference the new object, i.e. informing XCLE this object is in use. That, in turn, should be done using the **XCLE_ObjectUpRef** and **XCLE_ObjectDnRef** methods.

Referencing a new object (through **XCLE_ObjectUpRef**) is useless if its visibility will be restricted to the current block. It is recommanded whenever the object will persist after the end of the block. It is mandatory when the object will be duplicated, in part or whole (thing of the atoms in a list), in a container structure like **XCLE_Stack** or **XCLE_Hash**.

Destruction

The same holds for dereferencing an object (through **XCLE_ObjectDnRef**) that has not been created in this block, and has become useless. As well, this is mandatory for objects deleted from a container structure.

Methods and calls from the XCLE library are a special case: as integral to the memory management system, they always return objects in the same reference state as they were given (or unreferenced for new objects), unless specified otherwise.

When an object is not used anymore in the block, **XCLE_ObjectFree** (or a type-specific method) must be called on it. If the object is not owned at this time, it will be effectively freed.

Role of support structures

(De-)referencing is mandatory for operations on containers structures, but most operations on containers will take care for you of referencing inserted objects, and dereferencing objects deleted. Containers structures are not objects per-se, and thus have no reference counts of their own.

Freeing these structures will dereference (once only) and free their objects. Take special care when using methods that return an object from a container without deleting it. If you then free the container, or simply empty it, this object will have been dereferenced. If it is not used anywere else, this object will then have reached a zero reference count, and have been freed.

Parsing context

Parsing process

Defining primitive templates through Code_Def

Registering primitives sets

Execution context

Execution procedure

What are exceptions?

Raised exception stack

Common pitfalls

XCLE - Extending XCLE: primitives, modules, custom types.

Section

This tutorial focuses on ways to making XCLE handle more complex language constructs than the basic mecanism of scalars, lists and primitives. To fully understand the concepts we are contending with, you will need to be very familiar with the language structure, and XCL programmation. You will also need to have a good understanding of the reference counting mecanism, and at least a basic knowledge of the layout of the API.

Language extensions

The first way to extend XCLE is to provide new primitives. Actually, XCLE would not be very usefull without these "extensions". The default set of primitives, XCLstd, provides most of the basic ways data can be altered in XCLE, whether it is stored in a scalar or list type.
However, tailored use of the language often relies on ad-hoc handlers, references to external libraries, or optimized, computationally intensive algorithms. These can be integrated into XCLE through the definition of dedicated primitives, either defined through the C API, or loaded inside modules (for instance, XCLstd is one such module).

Modules and types

The second way provides new data types, by defining types as compound of existing types (making complex types usng lists is relatively easy), and utility primitives to handle them, that can be seen as constructor and methods.
We will see on an example that while this is relatively easy, these constructs are not always needed, and in any case rely heavily on the construction of new primitives. Thus you should be familiar with primitives coding before attempting to enact this part of the tutorial.

Writing primitives

Defining a primitive is essentially providing ways for your code to interact with the execution code of XCLE. That means how to pass arguments, how to return results, where are stored parameters, and identifiying the primitive.
The data needed by XCLE for this purpose is stored in the **XCLE_CodeDef** structure that you will learn to build step by step:

```
struct S_XCLE_CodeDef {
        char name[CDEF_NAME_LEN+1] ; // Primitive name
        unsigned char argc ; // Number of arguments
        XCLE_Type argt[CDEF_ARGS_LEN] ; // Types of required arguments (OR-ed type magic numbers)
        unsigned char retc ; // Number of results
        XCLE_Type rett[CDEF_ARGS_LEN] ; // Types of results
        XCLE_CodeOperator func ; // Machine code pointer
        char desc[CDEF_DESC_LEN+1] ; // Short description string
} ;
```

On the contrary, all you need to obtain is provided through the prototype of a C function, called "machine code", with the **XCLE_CodeOperator** predefined prototype:

```
static const XCLE_Exception _MACHINECODE_myprimitive
  (XCLE_ExecCtx ctx, XCLE_Stack stk, XCLE_Hash hsh, XCLE_Object dat)
```

This function will be specified in the structure above as the func pointer, and will be called each time XCLE needs to execute this primitive.

Prototypes

The first thing XCLE checks when executing a primitive, before even using the func pointer, is the prototype, that is, witch arguments are needed for this primitive, and what it will return. These are the argc/argt fields for the arguments, and the retc/rett fields for the results. argc/retc hold the number of items taken / returned, while argt/rett hold a mask describing the allowed types. This mask is built by OR-ing the magic numbers (the constants XCLE_INTG, XCLE_FLTP, ...) of the types that can be taken / returned from the stack. The index 0 in these arrays means the first level of the stack.

Primitive data, names and parameters

The name field of this structure is the name under witch the primitive will be displayed (as **my_primitive**). The desc field is a static string providing a brief, human-readable, description of the primitive's usage and arguments.

Machine code

This is the most important part of the definition: it is where actual action takes places, and where all non-straightforward checks (that is, other that argument-checking) must be done. It is also where you must take care of object references.

This machine code function takes several arguments: an execution context, an arguments stack, a variables hash, and a parameter object. You will need to use these to obtain arguments, returning the results your code produced, and signal errors.

Getting args

Arguments are already on the stack when the program enters the primitive's machine code. Thus, getting primitive arguments is simply a matter of taking objects from the **XCLE_Stack** argument of the function, with **XCLE_StackPop**.

As the object is stored in a variable, and will probably not be used anymore after the function exists, it must be freed (by **XCLE_ObjectFree**) before returning. Remember taht it can do no harm to free an object you don't use anymore in a primitive's machine code, since this object is either on the stack, and in this case its reference count is non-null, or really not owned, and therefore can be freed.

Returning results

As for arguments, results are simply put on the stack. Once the result objects are created, use **XCLE_StackPush** to put on the first level each result object in order.

The stack methods will take care of reference count for you.

Raising exceptions

The normal way of exiting from a primitive handler is to return the predefined OK exception **XCLE_EXCEPTION_OK**. Any other exception returned is considered an error code, and will halt execution of the program that contains this primitive.

In case of an error, use **XCLE_ExceptionNew** to create a new exception, and simply return it to raise the exception. Remember to free unused object before.

Standard error codes

The first argument to **XCLE_ExceptionNew** is an error code. A few predefined error codes exist, of the form ERR_<name>, with the corresponding MSG_<name> predefined error message.

Error name	Error number (#define ERR_<name>)	Error message (#define MSG_<name>)
OKSTAT	0	(no error)
UNHDLD	1	"Unhandled error"

RUNTIME	2	"Run time error"
MEMORY	3	"Memory error"
SYSTEM	4	"System error"
IOSTREAM	5	"IO error"
TOOFEWARG	6	"Too few arguments"
INVARGTYP	7	"Invalid argument type"
INVARGVAL	8	"Invalid argument value"
UNIMPLEM	9	"Not implemented"
NOSUCHVR	10	"No such variable"
OUTRANGE	11	"Value out of range"
PARSEERR	12	"Parse error"
USERERR	20	"User-defined error"

All values up to ERR_USERERR are reserved. User applications (i.e., your primitives) can define and use values bigger than ERR_USERERR for their own error system.

Modules

XCLE supports a dynamic primitive definition interface, through the use of precompiled modules. This let primitives sets to be distributed easily, and be used equally in a variety of interpreters.

Interface

XCL modules are shared objects (shared libraries, .so files on Linux, or .dll files on Windows) that define several symbols: **XCL_Registry_Table**, **XCL_Registry_Size** and **XCL_Registry_Vers**.

XCL_Registry_Table is a table of **XCLE_CodeDef** structures, that contains the list of primitives definitions. Each of these structures holds various information and data on a primitive, as we saw above.

XCL_Registry_Size is an integer holding the number of primitives we define (taht is, the size of **XCL_Registry_Table**

XCL_Registry_Vers contains a version identifier, that must match the current version of XCLE used. It should be built using the **XCLE_MAKEVERSIONID** macro, with three integer parameters: the major, minor and release numbers, as in **XCLE_MAKEVERSIONID(1,2,0)**

Each **XCLE_CodeDef** structure needs a function pointer, with type **XCLE_CodeOperator**. These functions must be defined in the same shared object as the symbols above, to complete the module’s symbol table and ensure proper loading.

Dependancy control

The **XCL_Registry_Vers** ensures that a module built for a given version of XCLE will not be loaded by an incompatible version of the library. The module loader will try to match one or several of the numbers given to **XCLE_MAKEVERSIONID** with an internal version identifier, and will fail with a warning when trying to load incompatible modules.

Implementation guidelines

Memory

While the reference counting system ensures that most of the burdensome memory management is dealt with by XCLE itself, using the library with C programs does not let us wrap away completely the problem. However, we tried to ensure that microscopic memory management needs to be done only when it makes the most sense, and where the scope of data is the easisest to interpret

Thus, the programmer’s part is reduced to freeing unused objects in primitives. Each argument that has been used and will be discarded, each temporary object not put on the stack, should be freeed (by a call to **XCLE_FreeObject**, or its type-specific counterpart) before exiting from the primitive handler.

Symbols

Symbol names (that is, function names) for the primitives machine code sections should be kept as distinctive of the module as possible (for instance, by prefixing them with the module name), to avoid symbol conflicts.
The use of global symbols, defined in the interpreter and not in the module, while unavoidable in some cases, should be kept to a minimum. The standard interpreters cxcl and gxcl define the **XCL_parse_ctx** and **XCL_exec_ctx** symbols (respectively, the parsing context and the execution context used by the interpreter, that are used by the standard modules in XCLstd.

Prototypes

More precise prototypes means more intensive use of the fast built-in argument checking system, and more precise results with the profiling tools. So you should to be as precise as possible when defining argument counts and types.

Example of module definition

Standard headers

These headers regroup the various XCLE components you will need, and define a few macros to ease primitive coding.

```
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

#include <DbgLog.h>

#include <XCLE/Containers.h>

#include <XCLE/Obj_generic.h>
#include <XCLE/Obj_void.h>
#include <XCLE/Obj_intg.h>
#include <XCLE/Obj_fltp.h>
#include <XCLE/Obj_strg.h>
#include <XCLE/Obj_list.h>
#include <XCLE/Obj_code.h>

#include <XCLE/Mod_string.h>
#include <XCLE/Mod_exec.h>
#include <XCLE/Mod_write.h>
#include <XCLE/Mod_parse.h>

#define XCLE_Primitive(name) \
        static const XCLE_Exception name(XCLE_ExecCtx ctx, XCLE_Stack stk, XCLE_Hash hsh, XCLE_Object dat)

#define GETSTACK() stk
#define GETNAMES() hsh
#define GETDATA() dat
#define SETSTATUS(e) return e
#define SETSTATUSMSG(m) return XCLE_ExceptionNew(ERR_MAXSYSERR,m)
#define SETSTATUSNUM(e) return XCLE_ExceptionNew(ERR_##e,MSG_##e)
#define STATUSOK() return XCLE_EXCEPTION_OK
```

Machine code definition

We define here only two primitives: one that adds two numbers, and another that flips the sign of a number. We distinguish the case where the numbers are integers, and floating-point values.

```
XCLE_Primitive(_CodeCall_numadd) {
        XCLE_Object obj = NULL ;
        XCLE_Object obj1 = NULL ;
        XCLE_Object obj2 = NULL ;
        obj1 = XCLE_StackPop(GETSTACK()) ;
        obj2 = XCLE_StackPop(GETSTACK()) ;
        switch(XCLE_ObjectType(obj1)) {
        case XCLE_OT_INTG:
                switch(XCLE_ObjectType(obj2)) {
                case XCLE_OT_INTG:
                        obj = XCLE_AnyToObject(XCLE_IntgNew( XCLE_IntgValue((XCLE_Intg)obj1) +
                        XCLE_IntgValue((XCLE_Intg)obj2) )) ;
```

```
                    break ;
                case XCLE_OT_FLTP:
                    obj = XCLE_AnyToObject(XCLE_FltpNew( XCLE_IntgValue((XCLE_Intg)obj1) +
                    XCLE_FltpValue((XCLE_Fltp)obj2) )) ;
                    break ;
                }
                break ;
        case XCLE_OT_FLTP:
                switch(XCLE_ObjectType(obj2)) {
                case XCLE_OT_INTG:
                    obj = XCLE_AnyToObject(XCLE_FltpNew( XCLE_FltpValue((XCLE_Fltp)obj1) +
                    XCLE_IntgValue((XCLE_Intg)obj2) )) ;
                    break ;
                case XCLE_OT_FLTP:
                    obj = XCLE_AnyToObject(XCLE_FltpNew( XCLE_FltpValue((XCLE_Fltp)obj1) +
                    XCLE_FltpValue((XCLE_Fltp)obj2) )) ;
                    break ;
                }
                break ;
        }
        XCLE_ObjectFree(obj1) ;
        XCLE_ObjectFree(obj2) ;
        if(obj==NULL) { SETSTATUSNUM(RUNTIME) ; /* return ; */ }
        if(XCLE_StackPush(GETSTACK(),obj)==-1) { SETSTATUSNUM(RUNTIME) ; /* return ; */ }
        STATUSOK() ;
        /* return ; */
}

XCLE_Primitive(_CodeCall_numneg) {
        XCLE_Object obj = NULL ;
        XCLE_Object obj1 = NULL ;
        obj1 = XCLE_StackPop(GETSTACK()) ;
        switch(XCLE_ObjectType(obj1)) {
        case XCLE_OT_INTG:
                obj = XCLE_AnyToObject(XCLE_IntgNew( - XCLE_IntgValue((XCLE_Intg)obj1) )) ;
                break ;
        case XCLE_OT_FLTP:
                obj = XCLE_AnyToObject(XCLE_FltpNew( - XCLE_FltpValue((XCLE_Fltp)obj1) )) ;
                break ;
        }
        XCLE_ObjectFree(obj1) ;
        if(obj==NULL) { SETSTATUSNUM(RUNTIME) ; /* return ; */ }
        if(XCLE_StackPush(GETSTACK(),obj)==-1) { SETSTATUSNUM(RUNTIME) ; /* return ; */ }
        STATUSOK() ;
        /* return ; */
}
```

Primitives definitions structures

We now need to implement the XCL module interface, taht is, to fill the required symbols with appropriate values. We begin by setting the version control identifier with the version number of the XCLE library we use for the build.

```
const unsigned long XCL_Registry_Vers = XCLE_MAKEVERSIONID(1,2,0) ;
```

We set **XCL_Registry_Size** with the number of defined primitives:

```
const unsigned long XCL_Registry_Size = 2 ;
```

And finally we fill the registry with the primitive definition data, specifying the primitives’ arity, expected arguments, name...

```
XCLE_CodeDef XCL_Registry_Table[2] = {
        {
                "+", // Name of the primitive
                2, { XCLE_OT_INTG|XCLE_OT_FLTP, XCLE_OT_INTG|XCLE_OT_FLTP }, // Number, and types, of needed
                arguments
                1, { XCLE_OT_INTG|XCLE_OT_FLTP }, // Number, and types, of result(s)
                (const XCLE_CodeOperator) _CodeCall_numadd, // Machine code pointer
                "Numeric addition" // Short description
        } ,
        {
                "-", // Name of the primitive
                1, { XCLE_OT_INTG|XCLE_OT_FLTP }, // Number, and types, of needed arguments
                1, { XCLE_OT_INTG|XCLE_OT_FLTP }, // Number, and types, of result(s)
                (const XCLE_CodeOperator) _CodeCall_numneg, // Machine code pointer
                "Numeric sign inversion" // Short description
        } ,
 };
```

Compiling

Once you have created, as above, your primitive definition file, you need to compile it to build a shared library. On Unices, assuming you have called that file my_primitives.c, this is easily done with:

```
$[shell]> cc -c my_primitives.c -o my_primitives.o
$[shell]> ld -shared -soname my_primitives.so my_primitives.o -o my_primitives.so
```

Custom data types

The <TYPE:data> construct

Defining the type handler

Commented examples

Getting arguments: <print>

Returning results and types: <inc>

Managing execution, exceptions: <loop>

The Matrix type

The <Matrix> built-in: type-name, and type checker
Handlers: overloading the <+>, <*>, </>, <^> operators