Implementation of Kyoto Common Lisp

Taiichi Yuasa and Masami Hagiya
Research Institute for Mathematical Sciences, Kyoto University

1. Introduction

Kyoto Common Lisp (KCL for short) is a full implementation of the Common Lisp language [1,2]. KCL is a highly portable Common Lisp system intended for several classes of machines, from mini/micro to mainframe. The key idea behind the portability is the use of the C language and its standard libraries as the interface with the underlying machines and operating systems: The kernel of the system is written in C and the rest of the system is written in Common Lisp. Even the compiler generates intermediate code in C. KCL is also an efficient and compact system: KCL regards the runtime efficiency of interpreted code as important as the efficiency of compiled code. The small size of the KCL system makes KCL suitable for the current computer technology, such as the use of virtual memory and cache memory.

Currently, there are major versions of KCL.

1. KCL/AOS running under Data General's original AOS/VS (Advanced Operating System / Virtual Storage) for Data General's Eclipse MV series super-minicomputers

2. KCL/VAX running under UNIX 4.2 bsd for Digital Equipment Corporation's VAX 11 series machines

3. KCL/SUN running under UNIX 4.2 bsd for Sun Microsystems' Sun 2 Workstation

4. KCL/UST running under UNIX V (Uniplus' version) for Sumitomo Electric Industries and Digital Computer Laboratory's personal workstation Ustation E15 (MC68000 base)

KCL/AOS is the original version of KCL, which was developed at Research Institute for Mathematical Sciences (RIMS), Kyoto University, with the cooperation of Nippon Data General Corporation. Other versions are ported from KCL/AOS at RIMS. All four versions share most of the source files of KCL. Improvements and error corrections are performed on the common source files, and the most recent revisions are brought to each machine from time to time. Ports to other machines and other operating systems are being undertaken or in preparation at other organizations.

2. Implementation Overview

The kernel of KCL is written in C, including:

* memory management and garbage collection
* the evaluator (or interpreter)
* Common Lisp special forms

All Common Lisp special forms are written in C. 418 Common Lisp functions and 11 Common Lisp macros are written in C, and the other Common Lisp functions and macros are written in Common Lisp. The KCL compiler is entirely written in Common Lisp.

The size of the source code is:

C code	705 Kbytes
Common Lisp functions and macros in Lisp	173 Kbytes
The compiler	264 Kbytes
total	1142 Kbytes

Three routines in the kernel are partly written in assembly language. These routines are:

* bignum multiplication
* bignum division
* bit table manipulation of the garbage collector

The total size of assembly code is 20 to 30 lines, depending on the version of KCL.

When KCL/AOS, the original version of KCL, was born, the following steps were taken to build it up.

1. Compile all C code with the C compiler and link them all. A subset of KCL is ready to run at this moment.

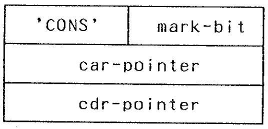
2. Load all Lisp code into KCL. Now the full system is ready to run, although the compiler and some Common Lisp functions and macros run interpretively.

3. Compile the source files of the KCL compiler with the (interpreted) KCL compiler itself. Load each Fasl-file (i.e., the file created by the KCL compiler) immediately after it is generated. The compilation process becomes faster toward the end of this step. Finally, the whole KCL compiler is ready to run by itself.

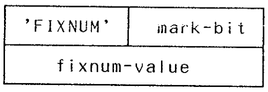
4. Compile the Common Lisp functions and macros written in Lisp, with the compiled KCL compiler. Load the Fasl-files. This completes the generation of the full system.

The same steps are taken whenever drastic changes are made to the kernel. On the other hand, the procedure to port KCL or to revise the ported versions of KCL is much simpler, because all Lisp code has been cross-compiled by the compiler of KCL/AOS beforehand.

KCL does not support the so-called immediate data. Any KCL object is represented as (a pointer to) a cell that is allocated on the heap. Each cell consists of several words (1 word = 32 bit) whose first word is in the format common to all data types: half of the word is the type indicator and the other half is used as the mark by the garbage collector. For instance, a cons cell consists of three words:



and a fixnum cell consists of two words:



Array headers and compiled-function headers are represented in this way, and array elements and compiled code are placed elsewhere.

Internally in compiled functions, certain Lisp objects may be represented simply by their values. For example, a fixnum object may be represented by its fixnum value, and a character object may be represented by its character code.

Cells of small fixnums ranging from -1024 to 1023 and cells of characters are pre-allocated in fixed locations. Thus, for example,

```
(eq 1023 1023)
```

yields t, whereas

```
(eq 1024 1024)
```

yields nil.

3. Memory Management

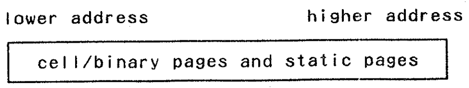
The whole heap of KCL is divided into pages (1 page = 2048 bytes). Each page falls in one of the following classes:

* pages that contain cells consisting of the same number of words
* pages that contain binary data such as compiled function code
* pages that contain relocatable data such as array elements

Free cells (i.e., those cells that are not used any more) consisting of the same number of words are linked together to form a free list. When a new cell is requested, the first cell in the free list (if it is not empty) is used and is removed from the list. If the free list is empty, then the garbage collector begins to run to collect unused cells. If the new free list is too short after the garbage collection, then new pages are allocated dynamically. Free binary data are also linked together in the order of the size so that, when a binary datum is being allocated on the heap, the smallest free area that is large enough to hold the binary datum will be used. Cell pages are never compactified. Once a page is allocated for cells with *n* words, the page is used for cells with *n* words only, even after all the cells in the page become garbage. The same rule holds for binary pages. In contrast, relocatable pages are sometimes compactified. That is, each relocatable datum may be moved to another place.

Fig.1 illustrates the actual configuration of the KCL heap There is a "hole" between the area for cell/binary pages and the area for relocatable pages. New pages are allocated in the hole for cell/binary pages, whereas new relocatable pages are allocated by expanding the heap to the higher address, i.e., to the right in the this figure. When the hole becomes empty, the area for relocatable pages are shifted to the right to reserve a certain number of pages as the hole. During this process, the relocatable data in the relocatable pages are compactified. No free list is maintained for relocatable data.

Symbol print names and string bodies are usually allocated in relocatable pages. However, when the KCL system is created, i.e., when the object module of KCL is created, such relocatable data are moved towards the area for cell/binary pages and then the pages for relocatable data are marked "static". The garbage collector never tries to sweep static pages. Thus, within the object module of KCL, the heap looks:



Notice that the hole is not included in the object module; it is allocated only when the KCL system is started. This saves the secondary storage a little bit. The maximum size of the hole is about 100 pages (= 200 Kbytes).

4. Stacks

KCL uses the following stacks.

* *Value Stack* for arguments/values passing, lexical variables allocation, and temporary values saving

* *Frame Stack* consisting of catch, block, tagbody frames

* *Bind Stack* for shallow binding of dynamic variables

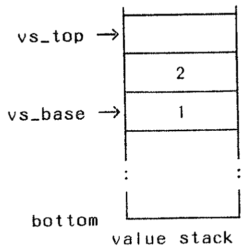
* *Invocation History Stack* maintaining information for debugging

* *C Language Control Stack*, sometimes used in compiled functions for arguments/values passing, typed lexical variables allocation, and temporary values saving, in addition to the obvious use such as function invocation

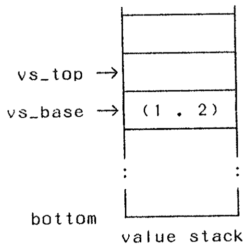
To show the argument/value passing mechanism, here we list the actual code for the Common Lisp function cons.

```
Lcons()
{
        object x;
        check_arg(2);
        x = alloc_object(t_cons);
        x->c.c_car = vs_base[0];
        x->c.c_cdr = vs_base[1];
        vs_base[0] = x;
        vs_pop;
}
```

We adopted the convention that the name of a function that implements a Common Lisp function begins with 'L', followed by the name of the Common Lisp function. (Strictly speaking, '-' and '*' in the Common Lisp function name are replaced by '_' and 'A', respectively, to obey the syntax of C.) Arguments to functions are pushed on the value stack. The stack pointer vs_base (value stack base) points to the first argument and another pointer vs_top points to the stack location next to the last argument. Thus, for example, when cons is called with the first argument 1 and the second argument 2, the value stack looks:



check_arg(2) in the code of Lcons checks if exactly two arguments are supplied to cons. That is, it checks whether the difference of vs_top and vs_base is 2, and if not, it causes an error. allocate_object(t_cons) allocates a cons cell in the heap and returns the pointer to the cell. After the car and the cdr fields of the cell are set, the cell pointer is put onto the value stack. The two stack pointers are used also on return from a function call. vs_base points to the first returned value and vs_top points to the stack location next to the last returned value. vs_pop in the code above decrements vs_top by one.



Because the same stack pointers are used both for argument passing and for return value passing, the Common Lisp function values does almost nothing. In most cases, the caller of a function uses only the first returned value which is pointed to by vs_base. This is not the case, however, when the called function returns no value at all. In order to avoid the check whether this is the case, each KCL function, on return from its call, sets nil to the stack entry which is pointed to by vs_base, whenever it returns no value at all. Thus, for instance, the actual code for the Common Lisp function values is:

```
Lvalues()
{
        vs_top[0] = Cnil;
}
```

where Cnil is a global variable that always contains the pointer to nil. See why this works.

5. The Interpreter

The KCL interpreter uses three A-lists (Association lists) to represent lexical environment: one for variable bindings, one for local function/macro definitions, and one for tag/block bindings. When a function closure is created, the current three A-lists are saved in the closure along with the lambda expression. Later, when the closure is invoked, the saved A-lists are used to recover the lexical environment.

6. The Compiler

The KCL compiler is essentially a translator from Common Lisp to C. Given a Lisp source file, the compiler first generates three intermediate files:

* a C-file which consists of the C version of the Lisp program

* an H-file which consists of declarations referenced in the C-file

* a Data-file which consists of Lisp data to be used at load time

The KCL compiler then invokes the C compiler to compile the C-file into an object file. Finally, the contents of the Data-file is appended to the object file to make a *Fasl-file*. The generated Fasl-file can be loaded into the KCL system by the Common Lisp function `load`. By default, the three intermediate files are deleted after the compilation, but, if asked, the compiler leaves them.

The merits of the use of C as the intermediate language are:

* The KCL compiler is highly portable. Indeed the four versions of KCL share the same compiler. Only the calling sequence of the C compiler and the handling of the intermediate files are different in these versions.

* Cross compilation is possible, because the contents of the intermediate files are common to all versions of KCL. For example, one can compile his or her Lisp program by the KCL compiler on Eclipse, bring the intermediate files to SUN, compile the C-file with the C compiler on SUN, and then append the Data-file to the object file. This procedure generates the Fasl-file for the KCL system on SUN. This kind of cross compilation makes it easier to port KCL.

* Hardware-dependent optimizations such as register allocations are done by the C compiler.

The demerits are:

* At those sites where no C compiler is available, the users cannot compile their Lisp programs.

* The compilation time takes long. 70% to 80% of the compilation time is used by the C compiler. The KCL compiler is perhaps the slowest compiler in the Lisp world.

The format of the intermediate C code generated by the KCL compiler is the same as the hand-coded C code of the KCL source programs. For example, supposing that the Lisp source file contains the following function definition:

```
    (defun add1 (x) (1+ x))
```

The compiler generates the following intermediate C code.

```
init_code(start,size,data)
char *start;int size;object data;
{       register object *base=vs_top;
        register object *sup=base+VM2;
        vs_check;
        Cstart=start;Csize=size;Cdata=data;
        set_VV(VV,VM1,data);
        MF(VV[0],L1,start,size,data);
        vs_top=vs_base=base;
}
/*      function definition for ADD1  */

static L1()
{       register object *base=vs_base;
        register object *sup=base+VM3;
        vs_reserve(VM3);
        check_arg(1);
        vs_top=sup;
        base[1]=one_plus(base[0]);
        vs_top=(vs_base=base+1)+1;
        return;
}
```

The C function `L1` implements the Lisp function `add1`. This relation is established by `MF` in the initialization function `init_code`, which is invoked at load time. There, the vector `VV` consists of Lisp objects; `VV[0]` in this example holds the Lisp symbol `add1`. `VM3` in the definition of `L1` is a C macro declared in the corresponding H-file. The actual value of `VM3` is the number of value stack locations used by `L1`, i.e., 2 in this example. Thus the macro definition

```
    #define VM3 2
```

is found in the H-file.

The KCL compiler takes two passes before it invokes the C compiler. The major role of the first pass is to detect function closures and to detect, for each function closure, those lexical objects (i.e., lexical variable, local function definitions, tas, and block-names) to be enclosed within the closure. This check must be done before the C code generation in the second pass, because lexical objects to be enclosed in function closures are treated in a different way from those not enclosed.

Ordinarily, lexical variables in a compiled function *f* are allocated on the value stack. However, if a lexical variable is to be enclosed in function closures, it is allocated on a list, called "environment list", which is local to *f*. In addition, one entity is reserved on the value stack, in which the pointer to the variable's location (within the environment list) is stored, so that the variable may be accessed by indexing rather than by list traversal. The environment list is a pushdown list: It is empty when *f* is called. An element is pushed on the environment list when a variable to be enclosed in closures is bound, and is poped when the binding is no more in effect. That is, at any moment during execution of *f*, the environment list contains those lexical variables whose binding is still in effect and which should be enclosed in closures. When a compiled closure is created during execution of *f*, the compiled code for the closure is coupled with the environment list at that moment to form the compiled closure. Later, when the compiled closure is invoked, as many entities as the elements in the environment list is reserved on the value stack, each of which points to a lexical object in the environment list, so that, again, each object may be referenced by indexing.

Let us see an example. Suppose the following function has been compiled.

```
(defun foo (x)
  (let ((a #'(lambda () (incf x)))
        (y x))
    (values a #'(lambda () (incf x y)))))
```

foo returns two compiled closures. The first closure increments x by one, whereas the second closure increments x by the initial value of x. Both closures return the incremented value of x.

```
>(multiple-value-setq (f g) (foo 10))
#<compiled-closure nil>

>(funcall f)
11

>(funcall g)
21

>
```

Fig.2 illustrates the status of the two compiled closures after these calls.

Declarations, especially type and function declarations, increase the efficiency of the compiled code. For example, for the following Lisp source file, with two Common Lisp declarations added,

```
(eval-when (compile)
  (proclaim
   '(function tak
      (fixnum fixnum fixnum) fixnum)))

(defun tak (x y z)
  (declare (fixnum x y z))
  (if (not (< y x))
      z
      (tak (tak (1- x) y z)
           (tak (1- y) z x)
           (tak (1- z) x y))))
```

the compiler generates the following C code.

```
/*  local entry for function TAK  */
static int LI2(V4,V5,V6)
int V4,V5,V6;
{  VMB3 VMS3 VMV3
   if((V5)<(V4)){
   goto T4;}
   VMR3(V6)
T4:;
   {int V7=LI2((V4)-1,V5,V6);
   {int V8=LI2((V5)-1,V6,V4);
   VMR3(LI2(V7,V8,LI2((V6)-1,V4,V5)))}}
}
/*  global entry for the function TAK  */
static L2()
{  register object *base=vs_base;
   base[0]=make_fixnum(LI2(fix(base[0]),
fix(base[1]), fix(base[2])));
   vs_base=base; vs_top=base+1;
}
```

The main part of the tak function is LI2. If redundant parentheses are removed, macros are expanded, and identifiers are renamed, we obtain the following code equivalent to LI2.

```
/*  local entry for function TAK  */
static int tak(x,y,z)
int x,y,z;
{
      if(y<x) goto L;
      return(z);
L:
   {    int t1=tak(x-1,y,z);
        int t2=tak(y-1,z,x);
        return(tak(t1,t2,tak(z-1,x,y)));
   }
}
```

This is almost hand-written tak code in C. The only overhead is the use of the temporary variables t1 and t2. This is necessary to make sure that the arguments are evaluated in the correct order (i.e., from left to right), since the C language does not specify the order of argument evaluation. If the compiler generated the following code,

```
return(tak(tak(x-1,y,z),
           tak(y-1,z,x),
           tak(z-1,x,y)));
```

the C compiler of Eclipse AOS/VS evaluates the three inner calls to tak from left to right (this is all right), whereas the C compiler of Unix evaluates from right to left (this is bad). In this example of tak, the order of evaluation does not matter actually, because tak causes no side effects. But the KCL compiler does not know that. The KCL compiler still has room for improvements.

7. Portability

Although KCL is made to be highly portable, certain minor changes had to be done, when it was ported to VAX Unix 4.2 bsd. These changes include:

1. The compiler top-level was slightly changed, because of the differences of the calling sequence of the C compiler and of the handling of object files.

2. File system interface was changed to fit Unix 4.2 bsd.

3. Machine-dependent Common Lisp system parameters, such as most-positive-short-float, and machine-dependent Common Lisp functions, such as decode-float, were redefined.

4. The three assembler routines were rewritten.

5. The in-core loader that loads Fasl-file into the KCL memory was changed. This was a simple job because we used the standard linkage editor ld of Unix.

6. The memory dump routine was rewritten using the standard system calls of Unix.

The whole job of poring KCL to VAX Unix 4.2 bsd took three days. Later, we spent some more days, to fix bugs in the ported version of KCL.

Port to SUN Workstation was much easier than port to VAX, mainly because the operating system is the same for both VAX and SUN. What has to be changed were:

1. The machine-dependent system parameters and functions were rewritten.

2. The three assembler routines were rewritten.

The whole job of poring KCL to SUN took three evenings. Most of the time was spent for the three assembler routines, because we did not know anything about the MC68000 assembler at first.

8. Evaluation

The size of the object module of the whole KCL system (including the Compiler) is:

KCL/AOS	1.78 Mbytes
KCL/VAX	1.45 Mbytes
KCL/SUN	1.56 Mbytes
KCL/UST	1.56 Mbytes

Since all system initialization (such as loading the database of the KCL compiler) has been done when the object module is created, the object module size roughly corresponds to the initial size of the KCL process when a KCL session is started, minus the initial size of the hole in the heap (about 200 Kbytes).

For the results of Lisp benchmark tests [3] with the four versions of KCL, refer to [4].

Acknowledgement: The authors wish to thank Dr. Reiji Nakajima.

References

[1] Steele, Guy L., *An Overview of Common LISP*, in *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, pp. 98-107, 1982.

[2] Steele, Guy L. *et.al.*, *Common Lisp: The Language*, Digital Press, 1984.

[3] Gabriel, Richard P. *Performance and Evaluation of Lisp Systems*, Computer Systems Ser. Research Reports, MIT Press, 1985.

[4] Yuasa, Taiichi and Hagiya, Masami, *Kyoto Common Lisp Report*, Teikoku Insatsu Publishing, 1985.

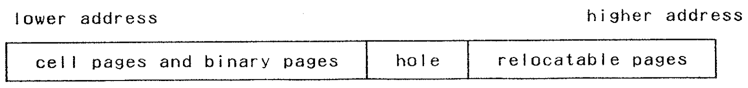


Fig.1. The Heap

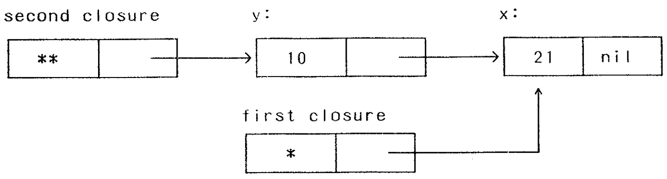


Fig.2. Lexical Environments in Compiled Closures
* : address of the compiled code for #'(lambda () (incf x))
** : address of the compiled code for #'(lambda () (incf x y))