Design and Implementation of Kyoto Common Lisp

Taiichi YUASA
Department of Information and Computer Sciences
Toyohashi University of Technology
Toyohashi 440, Japan

13 January 1990

Abstract

Kyoto Common Lisp (KCL for short) is a full implementation of the Common Lisp language. KCL is a highly portable Common Lisp system intended for several classes of machines, from workstations to mainframes. The key idea behind the portability is the use of the C language and its standard libraries as the interface to 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. This paper reports the implementation of KCL along with the design discussions to obtain a highly portable and yet efficient Lisp system.

1 Introduction

Kyoto Common Lisp (KCL for short) is a highly portable and efficient Lisp system based on the Common Lisp language as specified in [12]. The motivation of the development of KCL was to make a Lisp system available on Data General's MV series minicomputers, which the Research Institute for Mathematical Sciences (RIMS), Kyoto University decided to use as the main computer system. This decision was made in late 1982. Until then, we have been using DECSystem 2020 on which several Lisp systems were available including MacLisp [6,9], INTERLISP [13], and Standard Lisp [5]. Lisp was one of the major languages at RIMS and some projects there such as IOTA [7] highly depended on Lisp. On the other hand, since the MV had been used mainly in industrial world, no suitable Lisp system was available on the MV at that time [1] . Thus RIMS badly needed a Lisp system running on the new MV machines and a project was organized to develop a Lisp system on the MV before RIMS switched from DECSystem 2020 to the MVs in early 1984.

[1] It was in 1985 when Data General announced the availability of their own Common Lisp system.

The first job of the Lisp project was to choose a Lisp dialect suitable for our use. Since we did not have enough time to design our own dialect, we had to implement an already available dialect. None of the dialects at our hand were, however, completely satisfactory. INTERLISP was well documented and maintained, and equipped with powerful tools for program development. However, the memory space for user programs was relatively small and the performance of the system was not so good as other Lisp systems. In addition, the language was too large for us to implement in a short time. Standard Lisp was a compact and efficient system, but the language was too small for large-scale Lisp applications. MacLisp seemed a language of reasonable size, but the language specification was vague and unstable. Franz Lisp [2] was relatively satisfactory, but we had no running system at RIMS. Franz Lisp had a good manual, but since the manual was intended for the user but not for the implementor, it seemed quite difficult to implement Franz Lisp without referring to a running system.

It was early in 1983 when we were informed that the US activity for Lisp standardization was in progress and a preliminary document of Common Lisp was available for public review. The document we obtained [11] was still incomplete but contained enough information to determine that this dialect is suitable for our purpose for the following reasons.

- Common Lisp is a successor of MacLisp and thus is familiar to us.
- It seemed that if we had a Common Lisp system, we could easily exchange Lisp applications with other organizations.
- Common Lisp is relatively well-documented. It seemed that we could implement this language without referring to running systems. (Indeed, no running system of Common Lisp existed early in 1983.)
- Common Lisp has a rich set of features suitable for large-scale applications.
- Unlike most other Lisp dialects, Common Lisp is a lexically-scoped language. Lexical scope increases the reliability of application programs.

There were of course some drawbacks in Common Lisp.

- Common Lisp is too large to implement in a short time. Thus we decided to start with a reasonable subset and then to extend the system as we needed.
- The full set of Common Lisp requires a large memory space. Fortunately, however, memory chips were getting cheaper and most major operating systems were becoming to support a huge memory space. We decided not to worry too much about the problem of memory shortage.
- Common Lisp is a compiler-oriented language and the interpreter seemed less efficient than for other dialects. This was a severe problem because we first planned to implement only the interpreter. However, the chance was that once we had a compiler, the compiled code could run much faster than in other dialects.

The implementation was started in September 1983 and a subset implementation with only the interpreter was finished three months later. Then we began to write the compiler, as well as extending the system to support the full set Common Lisp. The first full set implementation with the compiler was finished in March 1984. This first version of KCL was much less efficient (both in memory and in runtime) than the current version. Major improvements were done during 1984 and 1985, after we obtained the final language specification of Common Lisp, the so-called CLtL [12].

As already mentioned, our motivation of the development of KCL was to make a Lisp system available on the new MV machines at RIMS. From the very beginning of the project, however, the portability of the system was the major concern to us. RIMS changes its main computer system every five years. If our system highly depended on the architecture of the MV, then we had to repeat our efforts when the MV was replaced by another machine. In addition, we were not familiar with the MV architecture. Rather than spending a lot of time to learn the details of the MV architecture, we chose to make the system independent of the architecture as much as possible.

In general, portability of a software system is proved only by actually porting the system to machines with different architecture and operating system. In order to prove the portability of KCL, we tried to port KCL onto two Unix BSD machines VAX and SUN, soon after we finished the first implementation of KCL. These porting efforts were quite successful. In either case, KCL began to run after three days of intensive work. We then tried to port KCL onto an MC68000 based Unix V workstation. This took a little more time because of the lack of some essential utilities we used to port KCL onto Unix BSD machines. These porting efforts of ours made KCL available on many other Unix machines as well. All versions of KCL share most of the source files, and the source files are organized so that it is clear which part of KCL is machine/OS dependent. Thus it seems relatively easy to port KCL onto a new machine at other organizations.

This paper overviews the design and implementation of KCL. Section 2 presents the overall structure of KCL. Sections 3 to 5 describe the data representation, the memory management, and the stacks of KCL to a certain degree. The interpreter and the compiler are then explained in Sections 6 and 7, respectively. Finally in Section 8, we discuss the portability of KCL.

2 An overview of the KCL implementation

There are three important design criteria to be considered by any Lisp implementors.

1. portability of the Lisp system
2. runtime efficiency
3. memory efficiency

These criteria are often in conflict and the order of precedence among the criteria strongly affects the design of the Lisp system. In the case of KCL, the portability is given the highest precedence for the reasons already mentioned. This means that KCL is not

always the most efficient Common Lisp system on a given machine. However, we believe our decision was correct. As the performance of computer systems has been increasing rapidly, the performance of the KCL system has been increasing "automatically". Rather than spending a lot of time to port the system onto a new, faster machine, we would like only to recompile the whole system for the machine. By reducing the time for the port, the life-time of the system on that machine becomes longer and the user can take advantage of the high performance of the machine before the machine becomes obsolete.

The second precedence is given to the runtime efficiency. As the cost of physical memory has been being reduced, popular machines are being equipped with more and more physical memory. It is not wise to reduce the runtime efficiency by too much worrying about the memory efficiency.

In order to obtain high portability of the system, we chose the C language as one of the implementation languages. The other implementation language is Common Lisp. The choice of the C language was based on the following considerations.

- The C language allows its compiler to generate more efficient code than other high level languages such as Pascal and PL/I.
- C language compilers are available on a wide spectrum of machines, from main frames to personal computers.
- Careful programming in the C language makes it possible to develop a highly portable system.
- Some Lisp systems such as Franz Lisp and InterLisp VAX [1] are written in the C language. We can take advantages of their experiences.

The "kernel" of KCL is written in C and the rest is written in Common Lisp. The kernel includes:

- memory management and garbage collection
- the interpreter
- Common Lisp special forms

In addition, about three fourth of the Common Lisp built-in functions and half of the Common Lisp built-in macros are written in C. The other Common Lisp functions and macros are written in Common Lisp. The KCL compiler is entirely written in Common Lisp.

A classical bootstrapping was used when the first version of KCL was built upon the MV machines.

1. All C codes were compiled by the C compiler and the objects were linked together. After this step, a subset of KCL became ready to run.
2. Then all Lisp codes (including those for the KCL compiler) were loaded into KCL. Now the full system became ready to run, although the compiler and some Common Lisp functions and macros ran interpretively.

3. The KCL compiler was incrementally compiled by the KCL compiler itself. Each time a compiler module was compiled, the object code of the module was loaded into KCL immediately. The compilation process became faster toward the end of this step. Finally, the whole KCL compiler was ready to run by itself.

4. Common Lisp functions and macros written in Lisp were compiled by the compiled KCL compiler. After the object code of these functions and macros had been loaded into KCL, the whole KCL system became ready to run.

The same steps were taken whenever drastic changes were 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. We will return to this issue in the description of the KCL compiler.

3 Data representation

Some versions of KCL does not support the so-called immediate data. In such versions, 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: one half of the word is the type indicator and the other half is used as the mark bit by the garbage collector. For instance, a cons cell consists of three words (see Figure 1 (a)) and a fixnum cell consists of two words (see Figure 1 (b)).

Objects of variable length such as arrays and compiled-functions are represented by a *header* and a *body*. The size of the header is fixed for each data type, and the format of the header is similar to fixed-length cells such as cons cells and fixnum cells. The variable-length part of the object is represented by the body which is pointed to by the header. For example, vectors (one-dimensional arrays) are represented as illustrated in Figure 2. By splitting the body from the header, access to the body requires pointer dereferencing and thus is slower than the alternative representation in which the header and the body are allocated in a contiguous location. However, this representation of KCL simplifies the memory compaction process during garbage collection and makes the garbage collector faster, as we will explain later.

Notice that only one bit is necessary as the mark bit. (This is why it is called "mark bit".) Also, since KCL has less than 30 implementation data types, only five bits are necessary for the tag field. Nevertheless, KCL uses a half word (16 bits) for each of these fields. This is one example of our design principle that runtime efficiency is more important than memory efficiency. Bit manipulations in the C language are sometimes very expensive and, much worse, are sometimes machine dependent. Thus we did not use compact representations for the mark bit and the tag field. Rather, we used standard notations in the C language to represent these fields. Here is the actual definitions of cons cells and vector headers in the C language.

```
struct cons {
    short   t, m;     /* tag field and mark bit */
    object  c_cdr;    /* cdr pointer */
```

```
    object  c_car;    /* car pointer */
};

struct vector {
    short   t, m;     /* tag field and mark bit */
    ...
    int     v_fillp;  /* length of the vector */
    object  *v_self;  /* pointer to the body */
    ...
};
```

The data type `object` in these definitions represents arbitrary Lisp objects in KCL, and is defined as follows.

```
typedef union lispunion *object;

union lispunion {
    ...
    struct cons     c;
    ...
    struct vector   v;
    ...
};
```

That is, each `object` is a pointer to the union that consists of all kinds of cells and headers. Incidentally, each kind of cells and headers is given a unique identifier, e.g., `c` for cons cells and `v` for vector headers. These identifiers are used to access a field of a cell or header. For example, if a variable `x` contains a pointer to a cons cell, the statement

```
x->c.c_car = y;
```

replaces the `car` field of the cell with the value of `y`.

Some versions of KCL support the so-called immediate data. If the machine supports byte addressing, then the two least significant bits (LSBs) of a cell pointer are always zero. This is because the number of bytes occupied by a KCL cell is always a multiple of four, and all cells are allocated from word boundaries. By using the two LSBs as object tags, we can represent three kinds of data objects as immediate data. The three kinds are fixnums, short-floats, and characters. Thus there are four kinds of Lisp objects in these versions of KCL as illustrated in Figure 3. Since the two LSBs are used as tags, fixnums are 30-bit signed integers and short-floats are 30-bit long. In order to obtain the actual fixnum values, we have to shift the fixnum object in two bits toward the LSB. As for short-floats, we do not care the actual representation, which depends on the machine. We only assume that

- short-floats are encoded in a 32-bit word,
- the two LSBs of the short-float representation are the least significant bits of the mantissa, and

- the two LSBs of short-float zeros and ones are zeros.

As far as we know, almost all floating point representations in use satisfies these assumptions. The last assumption is necessary because KCL uses floating point operations and comparisons provided by the machine (or by the C compiler). In order to get the actual short-float value, KCL just sets the LSB of the short-float object to zero. As for character objects, KCL just retrieves the higher half word to get the character code. In KCL, characters are represented by 16-bit code so that it can handle the Japanese character set consisting of more than 6000 characters.

4 Memory management

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

- *cell pages* that contain cells of fixed-size, such as cons cells and vector headers,
- *relocatable pages* that contain relocatable data such as vector bodies, and
- *binary pages* that contain binary data such as compiled function code

Free cells (i.e., those cells that are not yet used) 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 and binary pages are never compactified. Once a page is allocated for cells of n words, the page is used for cells of n words only, even after all the cells in the page become garbage. Similarly, once a page is used as a binary page, it is always used as a binary page.

The reason why cell pages and binary pages are never compactified is that it is very difficult to relocate objects in these area without the knowledge of the structure the C language control stack or registers. Since the kernel of KCL is written in C, cell pointers may be stored in C variables, which are usually allocated on the C stack or registers. If a cell were relocated, all C variables that contain a pointer to the cell had to be updated appropriately. However, where in the stack (or which register) a C variable is stored depends on the C language implementation. As for binary pages, the KCL compiler generates a portable C language code from a Common Lisp function definition, as will be explained later. The C language code includes calls to other C language functions. When the C language code calls another C language function, the return address is stored in the C stack or in one of the registers. If the code were relocated, such a return address had to be updated appropriately, but this process is also dependent on the implementation of the C language.

In contrast, relocatable pages are sometimes compactified. This is possible because data in relocatable pages (such as vector bodies) are pointed to only by the headers. By knowing which headers point to a datum in relocatable pages, KCL can relocate the datum without the knowledge of the C language implementation.

Figure 4 (a) illustrates the 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 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 these static pages. Thus, within the object module of KCL, the heap looks as in Figure 4 (b). 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. When the KCL system is started, it allocates a hole (about 100 pages, i.e., 200 Kilo bytes) toward the higher address of the heap. Then those cells and binary data produced by the user are allocated in the hole until the hole gets exhausted. Similarly, relocatable pages are allocated appropriately as the user requires.

The garbage collector in the current versions of KCL is a simple mark-and-sweep collector. Roughly speaking, the garbage collector runs in either of the two modes, *non-compactifying* and *compactifying*. In the non-compactifying mode, it first marks all used cells and binary data by recursively traversing object pointers. It then sweeps all cell/binary pages to collect free cells and binary data into the free lists. Garbages in the relocatable area are not collected in this mode. In the compactifying mode, the garbage collector collects garbages in the relocatable area by "shifting compaction". For this purpose, the collector first prepares an extra area toward the higher address of the heap, which is large enough to hold all data in the relocatable area. During the mark phase, each datum in the relocatable area that is pointed to by a used cell is copied into the extra area. The first such datum is copied into the lowest address of the extra area, the second into the next lowest address, and so on. Thus, after the mark phase, all used data in the relocatable area are stored in a contiguous memory space in the lower part of the extra area. The whole lower part is then moved towards the hole. This process effectively compactifies the whole relocatable area. Pointers to the relocatable data are updated during the mark phase, since the new address of each relocatable datum can be calculated by subtracting the size of the current relocatable area from the address of the extra area location into which the datum is copied.

5 Stacks

KCL uses the following stacks.

- *Value stack* for passing arguments to Lisp functions, returning values from Lisp functions, allocating lexical variables of compiled functions, and saving temporary values
- *Bind stack* for shallow binding of dynamic variables
- *Frame stack* consisting of catch, block, tagbody frames
- *Invocation history stack* maintaining information for debugging

In addition, KCL implicitly uses the control stack of the C language since the kernel and compiled functions are written in the C language.

The value stack is the "main stack" of the KCL system in the sense that all arguments to any Lisp function are passed via this stack and the return values are also passed via this stack. (The exception is that some compiled functions directly call other compiled functions if the compiler can arrange so. See Section 7.)

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 follow the syntax of the C language.) 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 as in Figure 5 (a). `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 signals an error. `allocate_object(t_cons)` allocates a cons cell in the heap and returns the pointer to the cell. (If no cell is available, the garbage collector will be invoked.) After the car and the cdr fields of the cell are set, the cell pointer is pushed 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. Thus, on return from the above call to `cons`, the value stack looks as in Figure 5 (b).

Because the same stack pointers are used both for argument passing and for return value passing, the Common Lisp function `values`, which receives any number of arguments and returns the nth argument as the nth value, 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 may return no value. Thus, for instance, the actual code for the Common Lisp function `values` is:

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

where `Cnil` represents the `nil` object.

In most Lisp implementations, the main stack is the source of the root pointers to start the marking phase of the garbage collection. This is also the case of the value stack of KCL. Unlike most other implementations, however, every entry of the value stack is a Lisp object in KCL. Other non-object information is stored in the other stacks. This simplifies the marking phase of the garbage collection and thus makes it faster.

As already mentioned, KCL is independent of the structure of the C language stack nor the available registers. This means that, any Lisp object currently in use must be pointed to directly or indirectly from some location other than the C language stack and registers. Thus we make it a rule that, whenever there is the possibility of garbage collection, i.e., whenever a new object is allocated, we use the value stack to save those objects that can otherwise be garbage collected. For example, if a built-in function written in the C language (or a compiled function) allocates two cons cells, it pushes the first cons cell onto the value stack before allocating the second cons cell. Such a protection of Lisp objects from the garbage collection seems to reduce the performance of the whole KCL a little bit, but not very much. To see how large is the overhead, we changed some built-in functions so that they do not protect Lisp objects, and compared the performance of the two versions of each such function. The result is that we could not see any difference in the performance of the two versions. Presumably this is because the process of cell allocation takes much more time than object protection, and thus the time for object protection is negligible.

KCL uses the so-called shallow-binding for dynamic variable bindings. The value of a dynamically bound variable is found in the *value slot* of the symbol cell that names the variable. When a variable is to be bound dynamically, the previous value of the value slot will be pushed onto the binding stack together with the name (i.e., the symbol that names the variable). At the end of the execution of the construct that binds the variable, the binding stack will be poped and the previous value will be restored into the value slot. It is necessary to push the variable name together with the previous dynamic value. This is because of the mechanisms of non-local exits of Common Lisp, such as `throw`ing to a `catch`er, `return`ing from a `block`, and `go`ing (jumping) to a tag in a `tagbody`. Whenever

these non-local exits occur, the binding stack must be "unwound" appropriately. That is, the binding stack will be poped an appropriate number of times, and each time the binding stack is poped, the poped value must be restored into the value slot of the appropriate symbol. The variable names pushed onto the binding stack are used to keep the symbol.

The frame stack consists of *frames*, each containing the status of the other KCL stacks at the time when the frame was pushed onto the frame stack. There are four kinds of frames corresponding to the Common Lisp special forms `catch`, `unwind-protect`, `block`, and `tagbody`. When the control enters a `catch` form, a *catch* frame will be pushed onto the frame stack, and so on. The frame is then used to recover the status of the other KCL stacks when a non-local exit occurs during the execution of the special form.

Another important information in a frame is the location to resume execution, after the non-local exit. In order to make it portable, KCL uses the standard mechanisms of the C language `setjmp` and `longjmp`. `setjmp` saves the location that immediately follows the call to the `setjmp` and the status of the C language stack into a structure called `jmpbuf`. By calling the `longjmp` with a `jmpbuf` structure as the argument, the control resumes from the point immediately after the corresponding call to `setjmp`. Thus, a non-local exit is implemented in KCL as the following steps.

1. Search the frame stack for an appropriate frame F.
2. If there is no appropriate frame, signal an error. The non-local exit is illegal, e.g, there is no `catch`er to `throw`.
3. Otherwise, pop the frame stack until the frame F is poped. This step actually pops frames one at a time. If an *unwind-protect* frame is poped, evaluate the expressions specified in the `unwind-protect` form that established the *unwind-protect* frame.
4. Reset the KCL stacks using the information stored in the frame and unwind the bind stack.
5. Call `longjmp`.

The invocation history stack is used for debugging. One typical use of this stack is to obtain the so-called backtrace. Usually, when a function is called from a user-defined function, a pair of the function name and the stack pointer to the value stack is pushed onto the invocation history stack. The debugger then uses this stack to obtain the arguments (which are pushed onto the value stack) to the function call. It is possible, however, to suppress such a pushing by setting the compiler switch when the user-defined function is compiled. Thus, once the user program has been debugged, the user can recompile the program so that nothing will be pushed onto the invocation history stack at runtime, in order to increase the performance of the program.

6 The Interpreter

The KCL interpreter uses three A-lists (Association lists) to represent lexical environment:

- One A-list is for variable bindings. Each element of the list is a pair. If the binding is lexical, the pair consists of the variable name and the current value of the variable. If the binding is dynamic, the pair consists of the variable name and the indicator of dynamic binding. In the latter case, the interpreter obtains the current value by looking up the value slot of the symbol that names the variable. When a variable is referenced from an interpreted code, if there is no pair with the specified variable name in this A-list, the reference is to a global variable, whose value is stored in the value slot of the symbol that names the variable.

- Another A-list is for local function/macro definitions. Each element of the list is a triple consisting of the name of the local function/macro, the indicator of whether this is for function or macro, and the function definition (or the macro expansion function). When a function or a macro is referenced from an interpreted code, if there is no pair with the specified function/macro name in this A-list, the reference is to a global function or macro, whose definition is stored in the *function slot* of the symbol that names the function/macro. Also, the symbol contains the indicator of whether a global function is defined under this name, a global macro is defined under this name, or no global function/macro is defined under this name.

- The last A-list is for tag/block bindings. Each element of the list is a triple consisting of the name of the tag/block, the indicator of whether this is for tag binding or block binding, and the "identifier" of the frame that was pushed onto the frame stack when the `tagbody` form or the `block` form was entered. When a `return-from` special form is evaluated in an interpreted code, if there is no pair with the specified block name in this A-list, the interpreter signals an error, and similarly for `go` special forms.

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.

Note that tag frames and block frames are identified by their identifiers, but not by their locations in the frame stack. This is because the frame may have already been discarded when the `go` or `return-form` is evaluated. Such a situation occurs in a call to a closure which tries to `return-from` a block or to `go` to a tag after the `block` or `tag` special form that established the frame is exited, as illustrated in the following example.

```
(defun foo ()
  (block bar
    #'(lambda () (return-from bar))))
```

This function `foo` establishes a block frame for the `bar` block and then returns with a function closure defined by the lambda expression. Before returning from `foo`, the `block` special form is exited and the corresponding block frame is poped. Thus, although the block is enclosed in the closure, the block frame does not exist on the frame stack any more. Therefore, it is an error to `return-from` the `bar` block in the closure, when the closure is called later. In order to handle such an illegal situation, the KCL interpreter searches the frame stack for a frame that has the specified frame identifier.

7 The Compiler

The KCL compiler is essentially a translator from Common Lisp to the C language (see Figure 6). 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 will be deleted after the compilation, but, if asked, the compiler will leave them. These three intermediate files can then be used to port the program onto another version of KCL running on a different machine. Contents of the three intermediate files are common to all versions of KCL. Thus, when porting an application program, the user need not recompile the Lisp source file. He can just recompile the intermediate C-file and build a Fasl-file. This strategy is also used whenever we port KCL onto a new machine. Remember that some part of KCL is written in Common Lisp. By using the intermediate C-files rather than the Lisp source files, the time required to port KCL is reduced to a great extent, because we can avoid the most time-consuming process to compile Lisp source files by the compiler that runs interpretively.

Common Lisp supports three built-in functions related with the compiler: `compile-file` which compiles a source file, `compile` which compiles an interpreted function already given to the system, and `disassemble` which shows the "disassembled" code of a given compiled function. `compile-file` is implemented in the way we have just explained. Since the C compiler is a file compiler, the built-in function `compile` in KCL first makes the intermediate files first. Then it calls the C compiler, builds a temporary Fasl-file, and then loads the Fasl-file. After the compiled function is installed, all the four temporary files are deleted. The last function `disassemble` causes a problem, since KCL has no direct relation with the assembler language of the operating system or the machine. The only purposes of `disassemble` are to help the implementor debug the Common Lisp system and to help the user increase the performance of his program. In the case of KCL, the best way to satisfy these purposes is to show the C language code generated from Lisp functions. However, it is impossible to reconstruct the C language code from the object code of compiled functions. Thus the function `disassemble` of KCL receives an interpreted function (or its name), translates it into the C code, and displays the C code on the standard output.

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

- The KCL compiler is highly portable. Indeed all 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, as we have already discussed above.
- Hardware-dependent optimizations such as register allocations are done by the C compiler.

The demerits are:

- If the user does not have a C compiler, he/she cannot compile his/her Common Lisp programs. Indeed, we received claims from a couple of users in the early days, when the C language was less popular than today.
- The compilation time takes long. 70% to 90% of the compilation time is used by the C compiler. The smaller the Lisp source program, the larger this ratio, because the time for the C compiler initialization is almost constant. As far as we know, the KCL compiler is the slowest Lisp compiler in the world.
- Any C compiler has its own limit on the maximum size of C source files. If the intermediate C-file becomes too large, the whole compilation fails. Since the limit is determined by many factors such as the number of identifiers, and since such a limit is not described in the manual of the C language implementation, there is no way for us to avoid the violation of the limit. We can only recommend the user to split his/her source file and recompile the smaller files.

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

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

the compiler generates the following 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 that appeared in the source file. `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 variables, local function definitions, tags, 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*. 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 the 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

>
```

In this example, `foo` is called with the argument `10`. It then returns two closures, which are assigned to the variables `f` and `g`. The first closure is then invoked and increments the lexical variable `x` by one. Next, the second closure is invoked and increments `x` by the value of the lexical variable `y`, which is `10` in this example. Since the value of `x` was incremented by the call of the first closure, the value of `x` becomes `21`. Figure 7 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 Lisp source file in Figure 8, which contains two standard declarations of Common Lisp (one given by the `proclaim` function and the other by the special form `declare`), the compiler generates the C code in Figure 9. The main part of the `tak` function is `LI2`. If redundant parentheses are removed, macros are expanded, and identifiers are renamed, we obtain the equivalent code in Figure 10. 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 be 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 order of the inner three calls to `tak` would be unpredictable. Indeed, the C compiler of the DG's MV series evaluates the three inner calls from left to right (this is all right), whereas the C compiler of Unix machines usually evaluates them from right to left (this is wrong). 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.

Let us explain more about the code in Figure 9. The C function `L2` implements the `tak` function. This function first converts the three fixnum arguments to integers of the C language, by using `fix`. It then calls the C function `LI2` which does most computation of the call to `tak`. When `LI2` is returned, `L2` converts the returned integer value into a fixnum value, by using `make_fixnum`. Then the fixnum value is pushed on the value stack and is returned as the value of the call to the `tak` function.

8 Portability

Although KCL has been proved to be highly portable, there are some implementation features of KCL that depend on the operating system and/or the machine. First of all, we assumed that the target machine of KCL has a 32-bit architecture. It is essential that pointers are 32-bit long. It would be practically impossible to port KCL onto machines without 32-bit architecture. The memory space is another obstacle to port KCL onto a small machine. The size of the object module of KCL is between 1.4 Mega bytes and 2 Mega bytes depending on the machine. Although the actual memory size required to run a user program depends on the program, at least 4 Mega bytes of logical memory space is expected to use KCL.

The following features of KCL are known to be machine/OS dependent.

- The compiler top-level slightly differs between versions of KCL, because of the difference of the calling sequence of the C compiler and of the handling of object files.
- The file system interface depends on the file system. For example, the algorithm for parsing file names differs between the versions on the MV and on Unix machines. Of course, the file system interface on Unix machines is the same.
- The in-core loader that loads a Fasl-file into the KCL heap differs between the versions on the MV and on Unix machines. For Unix machines, KCL uses the standard linkage editor `ld` of Unix. But if the machine does not support such a linkage editor, then one has to make it before porting KCL.
- The memory dump routine must be rewritten when porting KCL on a new non-Unix machine. Initialization of KCL (such as making built-in symbols and associating initial information with the symbols) takes some time (some minutes to some ten minutes). Rather than initializing KCL each time it is invoked, it is wise to save the memory image of KCL after the initialization and to invoke the memory image. For Unix machines, the memory dump routine to save the memory image is implemented with the standard system calls of Unix, and thus is common to all Unix machines.
- The procedure to install KCL differs. For Unix machines, this procedure is defined by the `make` facility and thus is common to all Unix machines. For other machines, we have to prepare a script for installation that is specialized for the machine.
- Some features of the Common Lisp language are machine/OS dependent. For instance, the value of the Common Lisp constant `most-positive-short-float` and the definition of the Common Lisp function `decode-float` depends on the representation of floating-point numbers. Much worse, some features of Common Lisp depend on the location where the Common Lisp system is used. For instance, the Common Lisp function `get-decoded-time` returns the time zone (`-9` for Japan), which depends on the country or on the area where the system is in use. The user himself has to modify such location-dependent features when he/she installs KCL (though we have never met a foreign user of KCL who has changed the time zone).

To sum up, it is straightforward to port KCL onto a new Unix machine. Often we only need to type `make` to the console to port to a new Unix machine. On the other hand, porting KCL onto a new non-Unix machine takes time, depending on the utility provided by the operating system. However, such a work is unavoidable to port any Lisp system, and we can say that the work is minimum in the case of KCL.

9 Summary and current work

We overviewed the design and implementation of the Kyoto Common Lisp system, which is highly portable, but also efficient. The use of the C language as the base implementation language is the key of the portability. Even the compiler generates C programs from Lisp programs. Thus no knowledge of the assembler language is necessary to port KCL. Although some implementation features of KCL is machine/OS dependent, these features are the minimum in that they are necessarily machine/OS dependent in any Lisp system. The portability of KCL among Unix machines is extremely high, and the performance of KCL is increasing each time a faster Unix machine appears.

The performance of the interpreter is not so good. This is partly because of the design of the Common Lisp language, which is a compiler-oriented language whereas most other Lisp dialects are interpreter-oriented. It seems that the KCL interpreter is a little bit slower than interpreters of other Common Lisp systems. This is mainly because we regarded the portability as the most important feature of KCL. Many machine/OS dependent improvements came to our mind but we discarded most of them simply because they would violate our design principle. On the other hand, the KCL compiler generates as efficient code as (or sometimes more efficient code than) commercial Common Lisp systems, although there are some overhead in KCL to maintain the portability. For the results of Lisp benchmark tests including the so-called Gabriel benchmarks [3], refer to [8,14].

There are two major improvements in the latest version of KCL. The one is the extension of the Common Lisp language to handle the Japanese character set, based on the proposal [4] of the SC22 Lisp Working Group of the Information Processing Society of Japan. The other improvement is the implementation of a real-time garbage collection algorithm designed by the author [15]. These improvements have already been finished and will become available to general users of KCL in the near future.

Acknowledgements

The development of KCL is a joint work with Masami Hagiya. The author wish to thank him especially for his great contribution to the high portability of KCL and for his porting work. The author also wish to thank Taichi Yasumoto for his implementation of immediate data and Japanese character set handling, and Toshiro Kijima for implementation of real-time garbage collection. Many other people helped us improve, port, maintain, and distribute KCL, but there are too many people to list all of their names here.

References

[1] Bates, Raymond; Dyer, David; and Koomen, Johannes. Implementation of InterLisp on the VAX. Conference Record of the 1982 LISP Conference, Pittsburgh, 1982.

[2] Foderaro, John K. and Sklower, Keith L. The Franz Lisp Manual. University of California Berkeley, 1982.

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

[4] Kurokawa, Toshiaki; Yuasa, Taiichi; Hashimoto, Yukiko; and Ito, Takayasu. Technical Issues on International Character Set Handling in Lisp. ISO/IEC JTC1/SC22/WG16 LISP, N33, 1988.

[5] Marti, J. B.; Hearn, A. C.; Griss, M. L.; and Griss, C. Standard LISP Report. University of Utah, 1978.

[6] Moon, David. MacLisp Reference Manual, Revision 0. MIT Project MAC, 1974.

[7] Nakajima, Reiji and Yuasa, Taiichi (ed). The IOTA Programming System, A Modular Programming Environment. Lecture Notes in Computer Science 160, Springer-Verlag, 1983.

[8] Okuno, Hiroshi G. (ed). The Report of The Third Lisp Contest and The First Prolog Contest. Report of WGSYM, No.33-4, IPSJ, 1985.

[9] Pitman, Kent. The Revised MacLisp Manual. MIT/LCR/TR 295, MIT Lab. for Computer Science, 1983.

[10] 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.

[11] Steele, Guy L. Common Lisp Reference Manual, Excelsior Edition. Carnegie-Mellon University, 1983.

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

[13] Teitelman, Warren et al. INTERLISP Reference Manual. Xerox Palo Alto Research Center, 1978.

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

[15] Yuasa, Taiichi. Real-time Garbage Collection on General-purpose Machines. Journals of Systems and Software, November, 1990 (to appear).

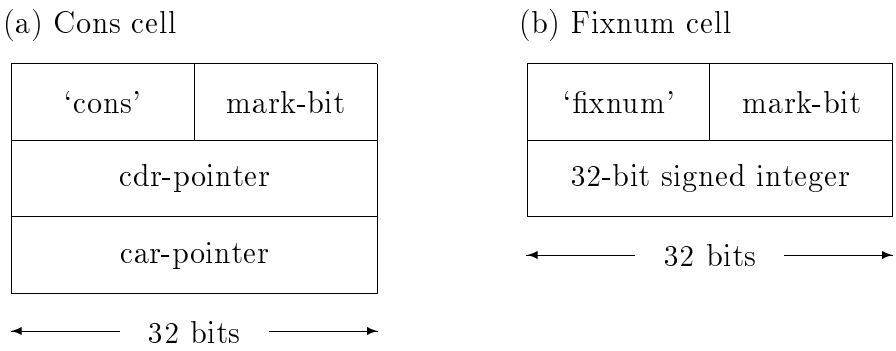


Fig.1. Cons cell and fixnum cell

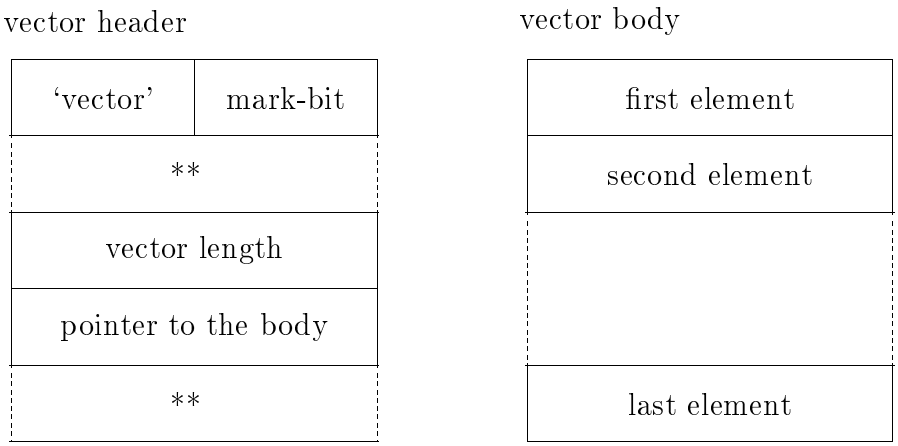


Fig.2. Vector header and vector body
('**' indicates that some other information is stored there.)

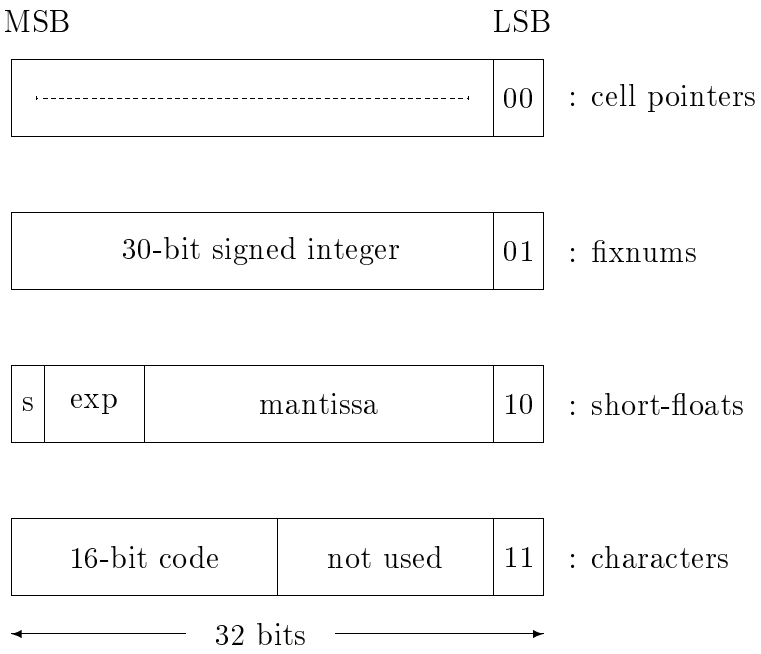
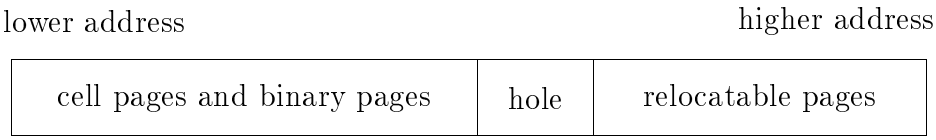


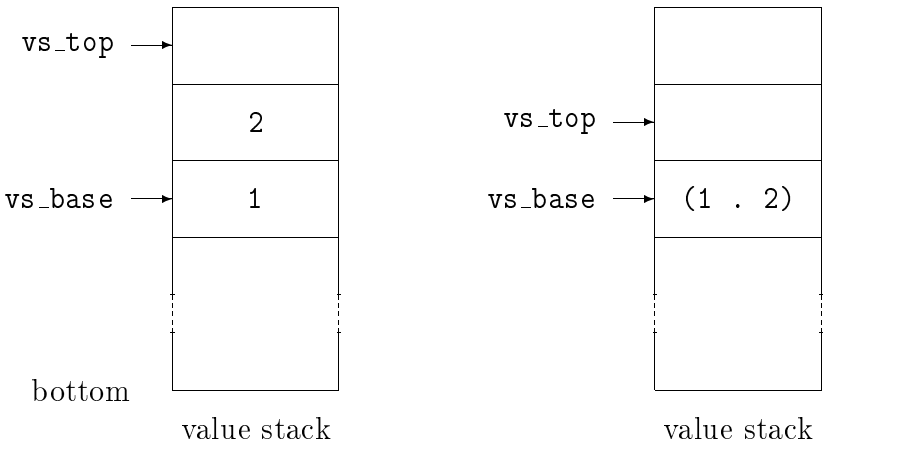
Fig.3. Four kinds of Lisp objects in KCL



(a) General configuration of the KCL heap

(b) Initial configuration of the KCL heap

Fig.4. The KCL heap



(a) When `cons` is called (b) When `cons` returns

Fig.5. The value stack

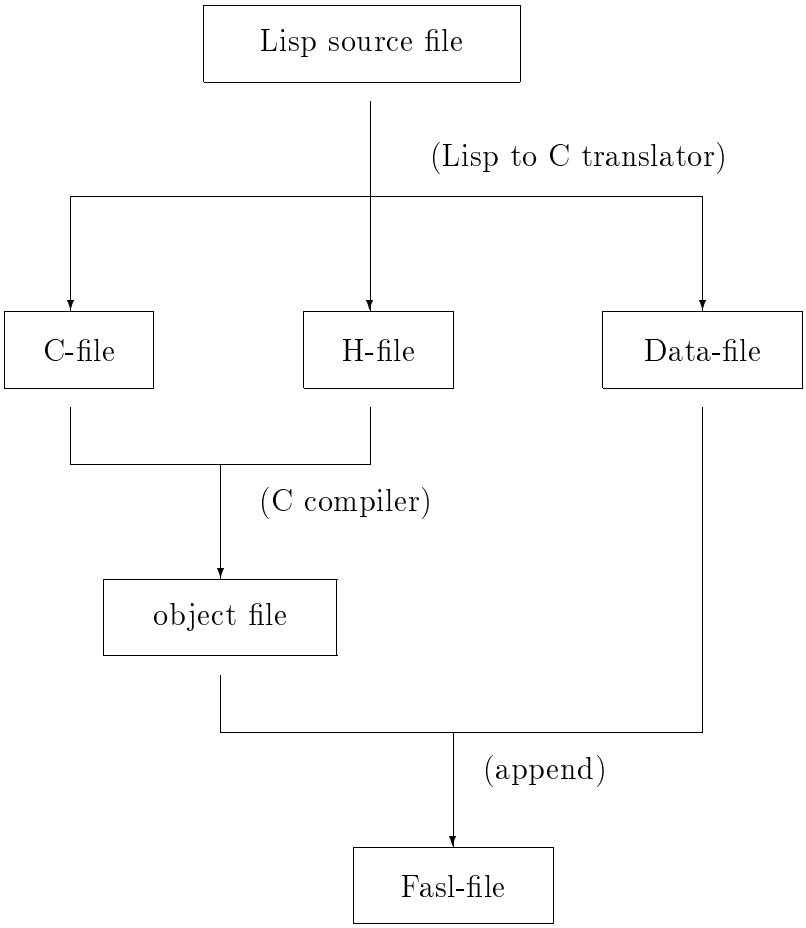


Fig.6. Data flow of the compiler

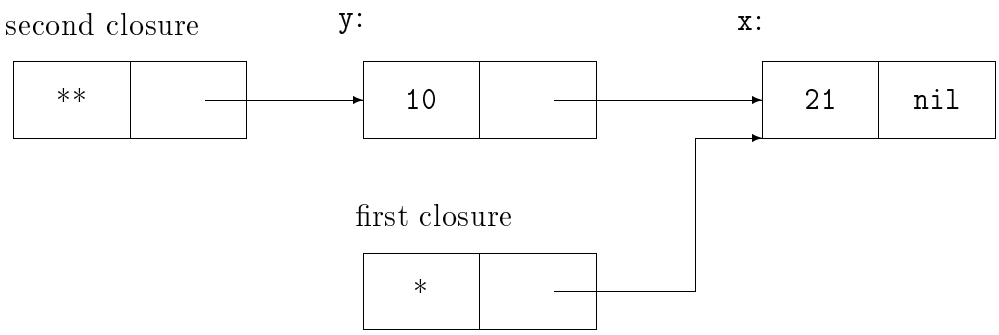


Fig.7. Lexical environments in compiled closures
* : address of the compiled code for `#'(lambda () (incf x))`
** : address of the compiled code for `#'(lambda () (incf x y))`

```
(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))))
```

Fig.8. A sample program with Common Lisp declarations

```
/*  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;
}
```

Fig.9. The actual C code for the program in Figure 8.

```
/*  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)));
    }
}
```

Fig.10. A code equivalent to the code in Figure 9.