Kyoto Common Lisp Report

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

Preface

Kyoto Common Lisp (KCL for short) is a full implementation of the Common Lisp language described in the Common Lisp Reference Manual:

Common Lisp: The Language
by Guy L. Steele et al.
Digital Press, 1984

All Common Lisp functions, macros, and special forms are defined in KCL, though a few of them have slightly different meanings from those described in the Common Lisp Reference Manual . All such differences are described in this report: If a Common Lisp function (or macro or special form) does not work as described in the Common Lisp Reference Manual and if this report does not describe the d ifference explicitly, then there must be a bug in KCL. All Common Lisp variables and constants are defined in KCL exactly as described in the Common Lisp Reference Manual .

Currently, there are four major versions of KCL:

1. KCL/AOS, under the AOS/VS operating system for Data General's ECLIPSE MV series machines.
2. KCL/VAX, under the Unix 4.2 bsd operating system for Digital Equipment Corporation's VAX 11 series machines.
3. KCL/SUN, under the Unix 4.2 bsd operating system for Sun Microsystems' Sun Workstation.
4. KCL/UST, under the Unix V (Uniplus' version) operating system for Sumitomo Electric Industries and Digital Computer Laboratory's Ustation E15.

KCL/ AOS is the original version of KCL, which was developed at the Research Institute for Mathematical Sciences (RIMS), Kyoto University, with the cooperation of Nippon Data General Corporation. The other three versions, which are collectively called *KCL on Unix*, are transplanted versions of KCL/AOS.

This report is intended to complement the *Common Lisp Reference Manual*. This report describes deviations of KCL from Common Lisp, those features specific to KCL, and the implementation-dependent functions of Common Lisp.

Acknowledgements

The project of KCL was supported by many people affiliated with many institutions. We are very grateful especially to the following people for their contributions to the KCL project.

First of all, we are grateful to the contributors to the design of Common Lisp.

Prof. Reiji Nakajima at RIMS, Kyoto University, provided us with considerable encouragement and moral support. Nippon Data General Corporation (NDG) helped us implement KCL/AOS. Mr. Teruo Yabe and Mr. Toshiyasu Harada joined us during the first stage of the KCL project and did a lot of coding. Mr. Takashi Suzuki and Mr. Kibo Kurokawa arranged the joint project. NDG is now supporting the distribution of KCL/AOS.

Data General Corporation in the United States sent us materials necessary to implement a Common Lisp system, such as the preliminary drafts of the Common Lisp reference manual and benchmark tests for Common Lisp. For the benchmark tests we are indebted to Dr. Richard Gabriel at Stanford University.

Dr. Daniel Weinreb at Symbolics answered most of our questions about the language specification. He also sent us the definition of rationalize written by Dr. Skef Wholey at CMU. We use this definition in KCL without any change.

Dr. Carl Hoffman at Symbolics checked the top-level of KCL and gave us advice for improving KCL. He also found some bugs in KCL and fixed them for us.

Mr. Naruhiko Kawamura at RIMS developed a Prolog system using the earliest version of KCL/AOS. That was one of the first big projects with KCL and he found many bugs.

Mr. Takashi Sakuragawa at RIMS hacked with KCL/AOS and gave us much advice concerning those features specific to KCL.

Mr. Tatsuya Hagino at Edinburgh University developed Micro EMACS on which FeCl2, the full-screen editor embedded in KCL/AOS, is based.

Mr. Kunihiko Nakamura at Kagawa University converted the assembly language version of Micro EMACS into the C language, which happened to become the prototype of FeCl2.

Prof. Akinori Yonezawa at Tokyo Institute of Technology encouraged us to port KCL/AOS to the VAX 11.

Mr. Etsuya Shibayama at Tokyo Institute of Technology helped us while we were working with the VAX 11 at the Institute.

Hagiwara Laboratory at Kyoto University offered (and is offering) their VAX 11 for finishing transplantation and maintaining KCL/VAX. We got also technical advice from people at Hagiwara Laboratory.

Prof. Shuji Doshita at Kyoto University offered the SUN Workstation at his Laboratory and gave us a lot of advice for transplantation to the SUN Workstation.

Mr. Takashi Hattori at RIMS gave us useful information about the Motorola 68000, the CPU chip of SUN Workstation.

Contents

Chapter 1

How to Start and End a KCL Session

KCL on Unix is invoked by the Shell command `kcl`.

```
% kcl
KCL (Kyoto Common Lisp)  July 1, 1985
```

————Note to KCL/AOS users————

KCL/AOS is invoked by the CLI command `KCL`.

```
) KCL
KCL (Kyoto Common Lisp) July 1, 1985
```

—————End of Note—————

When invoked, KCL will print the banner and initialize the system. The date in the KCL banner identifies the revision of KCL. `''July 1, 1985''` is the value of the function `lisp-implementation-version`.

If there exists a file named `init.lsp` in the current working directory, KCL successively evaluates the forms in the file, immediately after the system initialization. The user may set up his or her own KCL environment (e.g., the memory configuration) with `init.lsp`.

After the initialization, KCL enters the *top-level loop* and prints the prompt '>'.

```
>
```

The prompt indicates that KCL is now ready to receive a form from the terminal and to evaluate it.
Usually, the current package (i.e., the value of `*package*`) is the user package, and the prompt appears as above. If, however, the current package is other than the user package, then the prompt will be prefixed by the package name.

package-name>

To exit from KCL, call the function `bye` (or `by`).

```
>(bye)
Bye.
%
```

————Note to KCL/AOS users————

In this report, we sometimes assume that KCL is invoked from the Unix Shell. In particular, we use the standard prompt of the Unix Shell '`%`' in most examples.

—————End of Note—————

Alternatively, you may type `^D` (control-D), i.e., press the key D while pressing down the control key.

```
>^Dbye.
%
```

The top-level loop of KCL is almost the same as that defined in Section 20.2 of the *Common Lisp Reference Manual.* Since the input from the terminal is in line mode, each top-level form should be followed by a newline. If more than one value is returned by the evaluation of the top-level form, the values will be printed successively. If no value is returned, then nothing will be printed.

```
>(values 1 2)
1
2

>(values)

>
```

When an error is signalled, control will enter the break loop.

```
>(defun foo (x) (bar x))
foo

>(defun bar (y) (bee y y))
bar

>(foo 'lish)
Error: The function BEE is undefined.
Error signalled by BAR.

Broken at BAR.
>>
```

'>>' in the last line is the prompt of the break loop. Like in the top-level loop, the prompt will be prefixed by the current package name, if the current package is other than the `user` package.

To go back to the top-level loop, type `:q`

```
>>:q

Top level.
>
```

See Section 5.4 for the details of the break loop.

In KCL on Unix, the terminal interrupt (usually caused by typing `^C` (control-C) or by typing `DELETE`) is a kind of error. It breaks the running program and calls the break level loop.

Example:

```
>(defun foo () (do () (nil)))
foo

>(foo)
^C
Correctable error: Console interrupt.
Signalled by DO.
```

```
Broken at FOO.
>>
```

————Note to KCL/AOS users————

In KCL/AOS, the console interrupt caused by typing `^C` (control-C) followed by `^A` (control-A) is a kind of error. Typing `ctrlC` and `^A` breaks the running program and calls the break loop. On the other hand, the console interrupt caused by `^C` and `^B` (control-B) will immediately terminate KCL.

Example:

```
        >(defun foo () (do () (nil)))
        foo

        >(foo)
        ^C^A
        Correctable error: Console interrupt.
        Signalled by DO.

        Broken at FOO.
        >>(foo)
        ^C^B
*ABORT*
CONSOLE INTERRUPT
ERROR: FROM PROGRAM
        LEVEL 1
x,kcl
)
```

—————End of Note—————

Chapter 2

Data Types

KCL supports all Common Lisp data types exactly as defined in the *Common Lisp Reference Manual*. This chapter simply complements Chapter 2 of the *Common Lisp Reference Manual*, by describing implementation dependent features of Common Lisp data types. Each section in this chapter corresponds to the section in Chapter 2 of the *Common Lisp Reference Manual*, with the same section title.

2.1 Numbers

2.1.1 Integers

`Fixnums` in KCL are those integers in the range -(2 to the power of 31) to (2 to the power of 31)-1, inclusive. Other integers are `bignums`. Thus 25 factorial (25!)

```
15511210043330985984000000
```

is certainly a bignum in KCL.

Common Lisp constants related to integers have the following values in KCL.

```
most-positive-fixnum = 2147483647 = (2 to the power of 31)-1
most-negative-fixnum = -2147483648 = - (2 to the power of 31)

boole-1 = 3
boole-2 = 5
```

```
boole-and = 1
boole-andc1 = 4
boole-andc2 = 2
boole-c1 = 12
boole-c2 = 10
boole-clr = 0
boole-eqv = 9
boole-ior = 7
boole-nand = 14
boole-nor = 8
boole-orc1 = 13
boole-orc2 = 11
boole-set = 15
boole-xor = 6
```

See Chapter 12 of the *Common Lisp Reference Manual* for their meanings.

2.1.2 Ratios

There are no implementation-dependent features for ratios.

2.1.3 Floating-Point Numbers

KCL provides two distinct internal floating-point formats. One format is *short*; the other is *single* and serves also as *double* and *long*. The data types `single-float` , `double-float` , and `long-float` are considered to be identical, but `short-float` is distinct. An expression such as `(eql 1.0s0 1.0d0)` is false, but `(eql 1.0f0 1.0d0)` is true. Similarly, `(typep 1.0L0 'short-float)` is false, but `(typep 1.0L0 'single-float)` is true. For output purposes all floating-point numbers are assumed to be of *short* or *single* format.

The floating-point precisions are:

Format	KCL/AOS	KCL/VAX	KCL/SUN	KCL/UST
Short	24 bits	23 bits	24 bits	24 bits
Single	56 bits	55 bits	53 bits	53 bits
Double	56 bits	55 bits	53bits	53 bits
Long	56 bits	55 bits	53 bits	53 bits

The floating-point exponent sizes are:

Format	KCL/AOS	KCL/VAX	KCL/SUN	KCL/UST
Short	7 bits	8 bits	8 bits	8 bits
Single	7 bits	8 bits	11 bits	11 bits
Double	7 bits	8 bits	11 bits	11 bits
Long	7 bits	8 bits	11 bits	11 bits

There is no "minus zero." `(eql 0.0 -0.0)` is true.

Common Lisp constants related to floating-point numbers have the following values in KCL.

```
most-positive-short-float
  = - most-negative-short-float
  = 7.237005s75  (KCL/AOS)
    1.701412s38  (KCL/VAX)
    3.402823s38  (KCL/SUN and KCL/UST)

least-positive-short-float
  = - least-negative-short-float
  = 5.397605s-79  (KCL/AOS)
    2.938736s-39  (KCL/VAX)
    1.401298s-45  (KCL/SUN and KCL/UST)

most-positive-long-float
  = most-positive-double-float
  = most-positive-single-float
  = - most-negative-long-float
  = - most-negative-double-float
  = - most-negative-single-float
  = 7.237005577332264f75  (KCL/AOS)
    1.701411834604692f38  (KCL/VAX)
    1.797693134862315f308  (KCL/SUN and KCL/UST)

least-positive-long-float
  = least-positive-double-float
  = least-positive-single-float
  = - least-negative-long-float
  = - least-negative-double-float
  = - least-negative-single-float
  = 5.397605346934027f-79  (KCL/AOS)
```

```
    2.938735877055719f-39  (KCL/VAX)
    4.940656458412469f-324  (KCL/SUN and KCL/UST)

short-float-epsilon
  = 4.468372s-7  (KCL/AOS)
    6.938894s-18  (KCL/VAX)
    2.980232s-8  (KCL/SUN and KCL/UST)

short-float-negative-epsilon
  = 2.980232s-8  (KCL/AOS)
    6.938894s-18  (KCL/VAX)
    2.980232s-8  (KCL/SUN and KCL/UST)

long-float-epsilon
  = double-float-epsilon
  = single-float-epsilon
  = 1.110223024625157f-16  (KCL/AOS)
    6.938893903907228f-18  (KCL/VAX)
    5.5511151231257827f-17  (KCL/SUN and KCL/UST)

long-float-negative-epsilon
  = double-float-negative-epsilon
  = single-float-negative-epsilon
  = 6.938893903907228f-18  (KCL/AOS)
    6.938893903907228f-18  (KCL/VAX)
    5.5511151231257827f-17  (KCL/SUN and KCL/UST)

pi = 3.141592653589793
```

See Chapter 12 of the *Common Lisp Reference Manual* for their meanings.

2.1.4 Complex Numbers

There are no implementation-dependent features for complex numbers.

2.2 Characters

2.2.1 Standard Characters

KCL supports all standard and semi-standard characters listed in Section 2.2.1 of the *Common Lisp Reference Manual* . Non-printing characters have the following character codes.

```
Character        Code  (in octal)
-------------------------------
#\Space                 040
#\Newline               012
#\Backspace             010
#\Tab                   011
#\Linefeed              012
#\Page                  014
#\Return                015
#\Rubout                177
```

Note that `#\Linefeed` is synonymous with `#\Newline` and thus is a member of `standard-char`. Other semi-standard characters are not members of `standard-char`.

———— Note to KCL/AOS users————

KCL/AOS uses 025 (in octal) as the character code of `#\Backspace`.

————— End of Note —————

2.2.2 Line Divisions

Since KCL represents the `#\Newline` character by a single code 12 , problems with line divisions discussed in Section 2.2.2 of the *Common Lisp Reference Manual* cause no problem in KCL.

2.2.3 Non-standard Characters

KCL supports no additional non-standard characters.

2.2.4 Character Attributes

The bit and font fields of KCL characters are always 0 .

Common Lisp constants related to characters have the following values in KCL.

```
char-bits-limit = 1
char-code-limit = 256
char-control-bit = 0
```

```
char-font-limit = 1
char-hyper-bit = 0
char-meta-bit = 0
char-super-bit = 0
```

See Chapter 13 of the *Common Lisp Reference Manual* for their meanings.

2.2.5 String Characters

Since the bit and font fields of KCL characters are always 0 , `string-char` is considered to be identical to `character`.

2.3 Symbols

The print name of a symbol may consist of up to 16777216 (i.e., the value of `array-total-size-limit`) characters. However, when a symbol is read, the number of characters (not counting escape characters) in the print name is limited to 2048.

2.4 Lists and Conses

There are no implementation-dependent features for lists and conses.

2.5 Arrays

KCL arrays can have up to 64 ranks.

When the value of the Common Lisp variable `*print-array*` (see Section 22.1.6 of the *Common Lisp Reference Manual*) is `nil`, then bit-vectors are printed as `#<a bit-vector>`, other vectors are printed as `#<a vector>`, and other arrays are printed as `#<an array>`.

Common Lisp constants related to arrays have the following values in KCL.

```
array-dimension-limit = 16777216
array-rank-limit = 64
array-total-size-limit = 16777216
```

See Section 17.1 of the *Common Lisp Reference Manual* for their meanings.

2.5.1 Vectors

In KCL, array elements are represented in one of six ways depending on the type of the `array`.

Array Type	Element Representation
`(array t) and (vector t)`	`a cell pointer`
`(array fixnum) and (vector fixnum)`	`32 bit signed integer`
`(array string-char) and string`	`8 bit code`
`(array short-float) and (vector short-float)`	`32 bit floating point`
`(array long-float) and (vector long-float)`	`64 bit floating point`
`(array bit) and bit-vector`	`1 bit bit`

2.5.2 Strings

A string may consists of up to 16777216 (i.e., the value of `array-total-size-limit`) characters. However, when a string is read, the number of characters (not counting escape characters) in the string is limited to 2048.

2.5.3 Bit-Vectors

There are no implementation-dependent features for bit-vectors.

2.6 Hash Tables

All hash tables are printed as `#<a hash-table>`.

2.7 Readtables

All readtables are printed as `#<a readtable>`.

2.8 Packages

The following packages are built into KCL.

`lisp` `user` `keyword` `system` `compiler`

The `compiler` package contains symbols used by the KCL compiler. Other packages are described in Section 11.6 of the *Common Lisp Reference Manual.* The `system` package has two nicknames `sys` and `si` ; `system: symbol` may be written as `sys: symbol` or `si: symbol`. Other packages have no nicknames.

Packages are printed as `#<`*package-name* `package>`.

2.9 Pathnames

KCL provides a `#` macro `#"` that reads a pathname: `#"`*string*`"` is equivalent to `( apathname "`*string*`")` . For example,

```
#"foo.lsp"
```

is equivalent to

```
(pathname "foo.lsp").
```

The same format is used when a pathname is printed.

The initial value of the Common Lisp variable `default-pathname-defaults` is `#""` (or, equivalently, `(pathname "")`).

A pathname in the file system of Common Lisp consists of the following six elements:

```
host  device  directory  name  type  version
```

Among these elements, KCL does not use `host, device`, and `version`. That is, when converting a namestring into a pathname, KCL turns these three elements into `nil`. Conversely, when converting a pathname into a namestring, KCL ignores these three elements.

In the sequel, we explain how KCL converts a namestring into a pathname.

If a namestring contains one or more periods '.', the last period separates the namestring into the file name and the filetype.

```
"foo.lsp"
    name:          "foo"
    type:          "lsp"

"a.b.c"
    name:          "a.b"
    type:          "c"
```

If a namestring ends with a period, the filetype becomes the null string.

```
"foo."
     name:          "foo"
     type:          ""  (null string)
```

If a namestring begins with a period, the file name becomes `nil` .

```
".lsp"
     name:          nil
     type:          "lsp"
```

If a namestring contains no period, the filetype is `nil` .

```
"foo"
     name:          "foo"
     type:          nil
```

In a pathname, the file directory is represented as a list.

```
"common/demo/foo.lsp"
     directory:     ("common"   "demo")
     name:          "foo"
     type:          "lsp"
```

If a namestring does not contain a directory, the directory component of the pathname is `nil` .

```
"foo.lsp"
     directory:     nil
     name:          "foo"
     type:          "lsp"
```

In a pathname, the root directory is represented by the keyword `:root`.

```
"/usr/common/foo.lsp"
     directory:     (:root "usr" "common")
     name:          "foo"
     type:          "lsp"
```

The abbreviation symbols ' . ' and ' .. ' may be used in a namestring.

```
"./demo/queen.lsp"
     directory:     (:current\(ri ':current) "demo")
     name:          "queen"
     type:          "lsp"

"../../demo/queen.lsp"
     directory:     (:parent\(ri ':parent) :parent "demo")
     name:          "queen"
     type:          "lsp"
```

`:current` and `:parent` represent the current directory and the parent directory, respectively.

The part of a namestring after the last slash '/' is always regarded as representing the file name and the filetype. In order to represent a pathname with both the name and the filetype `nil`, end the pathname with a slash.

```
"/usr/common/"
     directory:     (:root "usr" "common")
     name:          nil
     type:          nil

"/usr/common/.lsp"
     directory:     (:root "usr" "common")
     name:          nil
     type:          "lsp"
```

' ∗ ' in the place of file name or filetype becomes :wild

```
"*.lsp"
     name:          :wild
     type:          "lsp"

"foo.*"
     name:          "foo"
     type:          :wild
```

———- Note to KCL/AOS users ————

In KCL/AOS, all the lower-case letters are turned into upper-case letters by pathname conversions. Thus, for example,

```
"foo.lsp"
     name:          "FOO"
     type:          "LSP"
```

KCL/AOS follows the convention of the AOS/VS file system: The symbols '/ ', ' . ', ' .. ', and ' ∗ ' in the examples above should be replaced by ' : ', ' = ', ' ^', and ' - ', respectively, in KCL/AOS.

————— End of Note —————

2.10 Streams

Streams are printed in the following formats.

#<input stream *file-name* >
An input stream from the file *file-name* .

#<output stream *file-name* >
An output stream to the file *file-name* .

#<string-input stream from *string* >
An input stream generated by (make-string-input-stream *string*).

#<a string-output stream>
An output stream generated by the function make-string-output-stream.

#<a two-way stream>
A stream generated by the function make-two-way-stream.

#<an echo stream>
A bidirectional stream generated by the function make-echo-stream.

#<synonym stream to *symbol* >
The stream generated by (make-synonym-stream *symbol*).

#<a concatenated stream>
An input stream generated by the function make-concatenated-stream.

#<a broadcast stream>
An output stream generated by the function make-broadcast-stream.

2.11 Random-States

KCL provides a # macro ' #$ ' that reads a random state. #$*integer* is equivalent to (make-random-state *integer*) . The same format is used when a random state is printed.

2.12 Structures

There are no implementation-dependent features for structures.

2.13 Functions

An interpreted function (including macro expansion functions) is represented in one of the following formats.
(lambda *lambda-list* . body)

A lambda-expression with null lexical environment and with no implicit block around it. This type of function typically appears when `'(lambda` *lambda-list* `. body )` is evaluated.

`(lambda-block` *block-name lambda-list* `. body )`

A lambda-expression with null lexical environment but with an implicit block around it. This type of function typically appears when `(defun` *function-name lambda-list* `. body )` is evaluated. In this case, *block-name* is identical to *function-name* .

`(lambda-closure` *env1 env2 env3 lambda-list* `. body )`

A lambda-expression with lexical environments but with no implicit block around it. This type of function typically appears when `#'(lambda` *lambda-list* `. body )` (or, equivalently, `(function (lambda` *lambda-list* `. body ))`) is evaluated. *env1 , env2 , and env3* represent the variable bindings, the local function/macro definitions, and the tag/block-name establishments, respectively, at the time the closure was created.

`(lambda-block-closure` *env1 env2 env3 block-name lambda-list* `. body )`

A lambda-expression with lexical environments and with an implicit block around it. Local functions and local macros are represented in this format. *env1 , env2* , and *env3* represent the variable bindings, the local function/macro bindings, and the tag/block-name establishments, respectively, at the time the local function/macro was created by `flet, labels`, or `macrolet`. The *block-name* is identical to the local function/macro name.

Compiled functions (including compiled macro-expansion functions) are printed in the following formats.

`#<compiled-function` *name* `>`

or

`#<compiled-closure nil>`

Incidentally, the value of `(symbol-function` *special-form-name* `)` is a list,

`(special .` *address* `)`

if *special-form-name* names a special form.

Common Lisp constants related to functions have the following values in KCL.

```
call-arguments-limit = 64
lambda-list-keywords = (&optional &rest &key &allow-
                        other-keys &aux &whole
                        &environment &body)
lambda-parameters-limit = 64
multiple-values-limit = 32
```

Refer to the *Common Lisp Reference Manual* for their meanings.

2.14 Unreadable Data Objects

There are no implementation-dependent features for unreadable data objects.

2.15 Overlap, Inclusion, and Disjointness of Types

In KCL, the types `number` and `array` are certainly subtypes of common, since KCL does not extend the set of objects of these types.

Chapter 3

Input and Output

3.1 Read Macros

The following # macros are introduced in KCL.

#" #" *string* " reads a pathname.
#" *string* " is equivalent to ((pathname " *string* ").

#$ #$*integer* reads a random state.
#$*integer* is equivalent to (make-random-state *integer*) .

The # macro '#,' works as described in the *Common Lisp Reference Manual* , only if it is included in a constant object. The forms immediately after '#,' below will be evaluated when the compiled code is loaded.

```
'#,x
'(a b c (d #,e f) g)
#(1 2 3 #,(+ a b c) 5 6)
#C(0.0 #,(exp 1))
```

Otherwise, the effect of using '#,' is unpredictable. Note that, when interpreted code is loaded, '#,' has the same effect as the # macro '#.'.

3.2 Input and Output Functions

The input and output functions of KCL almost follow the definitions in Chapter 22 of the *Common Lisp Reference Manual* . Most of the differences come from the fact that, in KCL, input from the terminal is always in line mode and binary I/O is not supported.

In KCL, `*terminal-io*` is a two-way stream from the standard input and to the standard output. The echoing to the terminal is performed by the underlying operating system. In particular, when a disk file is assigned to the standard output, nothing will be echoed at the terminal.

Those functions that deviate from the definitions in the *Common Lisp Reference Manual* are listed below.

`load` *pathname* `&key :print :verbose :if-does-not-exist` [*Function*]

If pathname does not specify the filetype of the input file, then load first tries to load a file with the filetype .o , i.e., the fasl file (see Chapter 6). If it fails, then `load` tries to load a file with the filetype `.lsp`. KCL assumes that `.lsp` is the standard filetype for source files. If it fails again, then `load` will load the specified file with no filetype.

`load` recognizes a file as a fasl file if and only if the filetype of the file is .o . Other files are assumed to be source files.

———— Note to KCL/AOS users ————-

In KCL/AOS, the filetype of fasl files is `.fasl` (see Chapter 6). Therefore, if the pathname argument does not explicitly specify the filetype, `load` first tries to load a file with the filetype `.fasl` . In addition, `load` recognizes a file as a fasl file if and only if the filetype of the file is `.fasl` .

—————- End of Note —————

`open` [*Function*]

The argument to the keyword variable `:element-type` and `:element-type` is always bound to the value `string-char` .

`close` [*Function*]

The keyword variable `:abort` is always ignored.

`listen` [*Function*]

`listen` always returns `t` .

`read-char-no-hang` [*Function*]

`read-char-no-hang` is equivalent to `read-char`

`clear-input` [*Function*]
`clear-output` [*Function*]

`clear-input` and `clear-output` simply return `nil` without doing anything.

`read-byte` [*Function*]
`write-byte` [*Function*]

These functions may operate on any stream. They read or write a byte (8 bits) at a time.

`princ` [*Function*]
`write-char` [*Function*]
`write-byte` [*Function*]

These functions do not always flush the stream. The stream is flushed when

1. a newline character is written, or
2. the input from the terminal is requested in the case that these functions operate on `*terminal-io*`

Chapter 4

Memory Management

4.1 Implementation Types

Each KCL object belongs to one of the 22 *implementation types*. The implementation types are shown in Table 4-1 with the corresponding Common Lisp data types. In the table, the compiled functions are divided into two implementation types; `cfun` is the type of compiled functions without environment, and `cclosure` is the type of compiled functions with environment (i.e., the type of compiled closures). `spice` is the type of internal data used by KCL, and does not correspond to any Common Lisp data type.

Table 4-1 Implementation Types

Implementation Type	*Common Lisp Data Type*
`cons`	`cons`
`fixnum`	`fixnum`
`bignum`	`bignum`
`ratio`	`ratio`
`short-float`	`short-float`
`long-float`	`long-float (= double-float = single-float)`
`complex`	`complex`
`character`	`character`
`symbol`	`symbol`
`package`	`package`
`hash-table`	`hash-table`
`array`	`(and array (not vector))`
`vector`	`(and vector (not string) (not bit-vector))`
`string`	`string`
`bit-vector`	`bit-vector`
`structure`	`structure`
`stream`	`stream`
`random-state`	`random-state`
`readtable`	`readtable`
`cfun`	`compiled-function without environment`
`cclosure`	`compiled-function with environment`
`spice`	`none`

Each object is represented by a cell allocated in the heap area of the interpreter. The size of the cell is determined by the implementation type of the object.

The implementation types are classified according to the size of the cells for the objects of the type, as shown in Table 4-2. The size of the cells in the same type class is the same.

Table 4-2 Classification of Implementation Types

Class	*Implementation Types*
1	`cons bignum ratio long-float complex`
2	`fixnum short-float character random-state readtable spice`
3	`symbol package`
4	`array hash-table vector bit-vector stream pathname cclosure`
5	`string cfun`
6	`structure`

For objects of the (implementation) types `readtable, symbol, package, array, hash-table, vector, bit-vector, stream, cclosure, string, cfun`, and `structure`, the cell is simply a header of the object. The body of the object is allocated separately from the cell and is managed in a different manner. The memory space occupied by the body of such an object is called a *block* . A block is either *contiguous* or *relocatable* depending on the area in which it is allocated. The difference between the two areas will be explained below. Table 4-3 lists these types, along with the contents of the body and the kind of the block.

Table 4-3 Types with Bodies

Type	*Body*	*Block*
`readtable`	`read table`	`contiguous`
`symbol`	`symbol name`	`relocatable`
`package`	`hash table`	`contiguous`
`array`	`array body`	`relocatable or contiguous`
`hash-table`	`hash table`	`relocatable`
`vector`	`vector body`	`relocatable or contiguous`
`bit-vector`	`bit-vector body`	`relocatable or contiguous`
`stream`	`I/O buffer`	`contiguous`
`cclosure`	`code`	`contiguous`
`string`	`string body`	`relocatable or contiguous`
`cfun`	`code`	`contiguous`
`structure`	`structure body`	`relocatable`

Usually, the body of an array, a vector, a bit-vector, or a string is allocated as a relocatable block. In KCL, the function `make-array` takes an extra keyword argument `:static`. If the `:static` argument is supplied with a non- `nil` value, then the body of the array is allocated as a contiguous block.

4.2 Heap and Relocatable Areas

The memory space of KCL is divided into two parts: the heap area and the relocatable area. Both areas occupy a contiguous space in the memory.

Cells of KCL objects are allocated in the heap. KCL divides the heap into pages (1 page = 2048 bytes), and each page consists of cells in the same type class (see Table 4-2). Cells in different type classes are allocated in different pages. Some blocks are also allocated in the heap: They are called contiguous blocks. The pages for contiguous blocks contain only contiguous blocks. Thus each page in the heap is either a page for cells in a particular type class, or a page for contiguous blocks. Blocks not in the heap are called relocatable blocks and are allocated in the relocatable area.

The user may specify the maximum number of pages that can be allocated for each type class by calling the KCL specific function `allocate`. There is also a limit on the number of pages for contiguous blocks; the limit can be altered by calling the KCL specific function `allocate-contiguous-pages` size of the relocatable area is specified by the KCL specific function `allocate-relocatable-pages`. See Section 4.4 for these functions.

In some installations of KCL, the total amount of memory that KCL can use is limited. In such cases, the entire memory may become exhausted before the maximum number of pages for each type class, for contiguous blocks, or for the relocatable area have been allocated.

The heap lies in a part of memory with lower address than the relocatable area and there is a "hole" between the two areas (see Figure 4-1). On request for a new page of heap, the page with the lowest address in the hole is used. When the hole is exhausted, the relocatable area is shifted toward the higher address space and a new hole of an appropriate size is created between the two areas.

Figure 4-1 Heap and Relocatable Area

```
lower address                                                  higher address
                  |       heap       |   hole   |relocatable area|
```

4.3 The Garbage Collector

The garbage collector of KCL has three levels according to what it collects:

1. cells
2. cells and relocatable blocks
3. cells, relocatable blocks and contiguous blocks.

In levels 2 and 3, the relocatable area is shifted to the higher address space to reserve an appropriate number of pages in the hole.

For each type class, KCL keeps a free list of unused cells, and when the free list is exhausted, a new page is allocated, or the garbage collector is invoked, depending on whether the maximum number of pages for that class have been allocated or not.

The garbage collector does not compactify the heap. That is, cells and contiguous blocks are never moved to another place. Moreover, once a page is allocated for a particular type class or for contiguous blocks, that page will never be freed for other classes, even if the entire page becomes garbage.

On the other hand, the relocatable area is compactified during level 2 and level 3 of garbage collection. A relocatable block is really relocatable.

The garbage collector is automatically invoked in one of the following situations. The number in the parentheses indicates the level of garbage collection that is performed.

* The free list of a certain type class is exhausted after the maximum number of pages have been allocated for that type class (1).
* The hole is exhausted (2).
* The relocatable area is exhausted after the maximum number of pages have been allocated for the relocatable area (2).
* The contiguous blocks are exhausted after the maximum number of pages have been allocated for contiguous blocks (3).

The garbage collector is also invoked by the following KCL specific function.

`gbc x` [*Function*]

The garbage collector is invoked with the level specified by `x`. If `x` is `nil`, the garbage collector is invoked for level 1 garbage collection. If `x` is `t`, it is invoked for level 3 garbage collection. Otherwise, it is invoked for level 2 garbage collection.

4.4 Allocation Functions

The following functions are used to set or inspect the (maximum) number of pages for each type class, for contiguous blocks, or for relocatable blocks.

`allocate` *type number* [*Function*]

Sets the maximum number of pages for the type class of the implementation type *type* to *number*. If more than *number* pages have already been allocated, an error is signalled.

`allocated-pages` *type* [*Function*]

Returns the number of pages currently allocated for the type class of the implementation type *type*.

`maximum-allocatable-pages` *type* [*Function*]

Returns the current maximum number of pages for type class of the implementation type *type*.

`allocate-contiguous-pages` *number* [*Function*]

Sets the maximum number of pages for contiguous blocks to *number*.

`allocated-contiguous-pages` [*Function*]

Returns the number of pages allocated for contiguous blocks.

`maximum-contiguous-pages` [*Function*]

Returns the current maximum number of pages for contiguous blocks.

`allocate-relocatable-pages` *number* [*Function*]

Sets the maximum number of pages for relocatable blocks to *number*. The relocatable area is expanded to *number* pages immediately. Therefore, "the current maximum number" and "the number of pages allocated" have the same meanings for relocatable blocks.

`allocated-relocatable-pages` [*Function*]

Returns the number of pages allocated for relocatable blocks.

If the pages for a particular type class are exhausted after the maximum number of pages for that class have been allocated, and if there remain no free cells (actually, if there remain very few cells), KCL behaves as directed by the value of the KCL specific variable `*ignore-maximum-pages*`. If the value is `nil`, then KCL signals a correctable error and enters the break loop. The user can reset the maximum number by calling `allocate` and then continue the execution of the program by typing `:r`.

Example:

```
>(make-list 100000)

Correctable error: The storage for CONS is exhausted.
                   Currently, 531 pages are allocated.
                   Use ALLOCATE to expand the space.
Signalled by MAKE-LIST.

Broken at FUNCALL.
>>(ALLOCATE 'CONS 1000)
t

>>:r

(nil nil nil nil nil nil nil nil nil nil ...........
```

The user can also reset the maximum number of pages for relocatable blocks and for contiguous blocks in a similar manner. On the other hand, if the value of `*ignore-maximum-pages*` is non- `nil`, then KCL automatically increments the maximum number of pages for the class by 50 percent. The initial value of `*ignore-maximum-pages*` is `t`.

4.5 Storage Information

`room & optional` *x* [*Function*]

The function `room` prints the storage information. The argument *x* is simply ignored and the output of `room` is always in the same format. `room` prints the following information:

```
* for each type class
    * the number of pages so-far allocated for the type class
    * the maximum number of pages for the type class
    * the percentage of used cells to cells so-far allocated
    * the number of times the garbage collector has been
      called to collect cells of the type class
    * the implementation types that belong to the type class
* the number of pages actually allocated for contiguous blocks
* the maximum number of pages for contiguous blocks
* the number of times the garbage collector has been called to
  collect contiguous blocks
* the number of pages in the hole
* the maximum number of pages for relocatable blocks
* the number of times the garbage collector has been called to
  collect relocatable blocks
* the total number of pages allocated for cells
* the total number of pages allocated
* the number of available pages
* the number of pages KCL can use.
```

The number of times the garbage collector has been called is not shown, if the number is zero.

In the following example, the maximum of `531` pages have already been allocated for the type class to which cons belongs, but only 16.9 percent of the cells are actually used. The garbage collector was once invoked to collect cells in this type class.

```
>(room)
 531/531    16.9%  1  cons bignum ratio long-float complex
   3/52     10.4%     fixnum short-float character random-state
                      readtable spice
  47/65     73.6%     symbol package
   3/71     32.4%     array hash-table vector bit-vector stream
                      pathname cclosure
  46/96     98.8%     string cfun
   1/32      2.3%     structure

  17/512              contiguous (3 blocks)
      14              hole
      50    47.4%  2  relocatable

   631 pages for cells
   712 total pages
 14840 pages available
 16384 maximum pages

 >
```

Chapter 5

Debugging Facilities

5.1 The Tracer

The Tracer causes selected `functions` to be traced. When such a traced `function` is invoked, it prints

level > (*name arg1 ... argn*)

On return from a traced `function`, it prints

< *level* (*name value1 ... valuen*)

name is the name of the traced `function`, *args* are the arguments, and *values* are the return values. *level* is a number which is incremented each time a traced `function` is invoked and is decremented at the completion of the invocation. Trace print-outs are indented according to the *level*.

In the current version of KCL, macros and special forms cannot be traced.

`trace` {*function-name*}* [*Macro*]

Causes one or more functions to be traced. *function-names* must be symbols and they are not evaluated. If a function is called from a compiled function, the call may not produce trace print-outs. If this is the case, the simplest way to get trace print-outs is to recompile the caller with a `notinline` declaration for the called function (see Chapter 7). `trace` returns a name list of those functions that were traced by the call to trace . If no *function-name* is given, `trace` simply returns a name list of all the currently traced functions.

`untrace` {*function-name*}* [*Macro*]

Causes the specified functions to be not traced any more. *function-names* must be symbols and they are not evaluated. `untrace` returns a name list of those functions that were untraced by the call to `untrace`. If no *function-name* is given, `untrace` will untrace all the currently traced functions and will return a list of their names.

5.2 The Stepper

`step` *form* [*Macro*]

Starts evaluating the it form in the single-step mode. In this mode, before any form is evaluated, the Stepper will print the form and prompt the user for a Stepper command. The Stepper binds the two variables `*print-level*` and `*print-length*` both to `2`, so that the current form may not occupy too much space on the screen. A Stepper command will be executed when the user types the single character for the command followed by the required arguments, if any, and presses the newline key. If the user presses the newline key without having typed any character, then the Stepper will assume that the Stepper command `n` was abbreviated.

The Stepper commands are:

`n` Next. Evaluates the current form in the single-step mode.

`s` Skip. Evaluates the current form in the ordinary mode. The single-step mode will be resumed at completion of the evaluation.

`p` Print. Pretty-prints the current form and then prompts again.

`f`*fn* Function. Evaluates the current form in the ordinary mode until the specified function *fn* is invoked. If the specified function is not invoked at all, then this command has the same effects as the q command below.

`q` Quit. Evaluates the current form and any other forms in the ordinary mode.

`e`*form* Eval. Evaluates the specified form in the ordinary mode and prints the resulting values. Then prompts again with the same current form.

`?` Help. Lists the Stepper commands.

5.3 Errors

`*break-enable*` [*Variable*]

This variable is used to determine whether to enter the break loop (see Section 5.4) when an error occurs. Even the function `break` checks this variable. Initially, this variable is set to `t`, and thus an error will invoke the break loop. If the value is `nil`, functions that cause fatal errors, such as `error`, will just print an error message and control will return to the top-level loop (or to the current break loop, if already in the break loop). Functions that cause correctable errors, such as `cerror`, will print an error message and a "continue message", and control will return to the next form. In KCL, backtrace is not part of an error message, but a break loop command will print backtrace. Therefore, if `*break-enable*` is `nil`, no backtrace appears on the screen.

When the break loop is entered, `*break-enable*` will be bound to `nil`.

5.4 The Break Loop

The break loop is a read-eval-print loop similar to the top-level loop. In addition to ordinary Lisp forms, the break loop accepts various commands with which the user can inspect and modify the state of the program execution. Each break loop command is identified with a keyword (i.e., a symbol in the `keyword` package). A break loop command is executed when the user inputs a list whose first element is the keyword that identifies the command. The rest of the list is the arguments to the command. They are evaluated before being passed to the command. If the command needs no arguments, then the user may input only the keyword. It is an error if the given keyword does not identify any command. Any other input to the break loop is regarded as an ordinary Lisp form; the form will be evaluated and the resulting values will be printed on the terminal.

There can be several instances of the break loop at the same time, and each such instance is identified by a *level number*. When the break loop is entered during execution in the top-level loop, the break loop instance is given the level number 1 . The break loop instance that is entered from the level n break loop is given the level number $n+1$. The prompt of the level n break loop is $n+1$ consecutive $>$'s, occasionally prefixed with the name of the current package.

The break loop keeps track of the invocation sequence of functions (including special forms and macro expansion functions), which led up to the break loop from the previous break loop (or from the top-level loop, if the current break loop is level 1). The invocation sequence is maintained in a pushdown stack of *events*. An event consists of an *event function* and an *event environment*. An event function is:

1. an interpreted (i.e., not compiled) function (global function, local function, lambda-expression, or closure),
2. a special form within an interpreted function,

3. a macro expansion function called from an interpreted function,

4. a compiled function called from an interpreted function, or

5. a compiled function called from another compiled function which was compiled while the `safety` optimize level is 3 or with a `notinline` declaration for the called function (see Chapter 7).

An event is pushed on the event stack when execution of its event function begins, and is poped away at the completion of the execution. An event environment is the 'environment' of the event function at the time the next event is pushed. Actually, an event environment is a pointer to the main stack of KCL. For each interpreted event function (i.e., event function in classes 1, 2, and 3), the pointer points to the first entry of the three contiguous main stack entries that hold the lexical environment of the event function. For each compiled event function (i.e., event function in classes 4 and 5), the pointer is set to the first entry of the main stack area that is used locally by the compiled code. In most cases, the first argument to the compiled function is saved in the first entry, the second argument in the second entry, and so on. The local variables of the function are allocated in the entries following the arguments. However, this is not always the case. Refer to Section 7.3 for variable allocations in compiled functions.

By break level commands, the user can choose one of the events as the *current event*. If the current event function is an interpreted event function, then the break loop evaluates Lisp forms in the lexical environment retrieved from the event environment. In particular, local variables may be referenced by the variable names, local functions and local macros may be invoked as usual, established blocks may be exited from, and tags may be used as the destination of `go`. If the current function is a compiled function, Lisp forms are evaluated in the null environment.

Within the break loop, each event is represented by the *event symbol*. The `:backtrace` command, for example, lists events in terms of their event symbols. If the event function is a named function (global or local) or a macro expansion function, then the function or macro name is used as the event symbol. If the event function is a special form, then the name of the special form is used. If the event function is a lambda-expression (or a closure), then the symbol lambda (or lambda-closure) is used.

To suppress unnecessary information, the user can hide (or make invisible) some of the events. Invisible events do not appear in the backtrace, for example. Initially, only those events are invisible whose event symbols belong to the system internal package system. When the break loop is entered, the last visible event becomes the current event.

The break loop commands are described below. Some of the commands allow abbreviation in the keywords that identify them. For example, the user may abbreviate `:current` as `:c`. The break loop commands return no values at all.

`:current` [*Break Loop Command*]
`:c` [*Abbreviated Break Loop Command*]

Prints the event symbol of the current event.

`:previous &optional` *n* [*Break Loop Command*]
`:p &optional` *n* [*Abbreviated Break Loop Command*]

Makes the *n*-th previous visible event the new current event. Invisible events are not counted. If there are less than *n* previous events, then the first visible event in the invocation sequence becomes the new current event. *n* must be a positive integer and the default is `1`.

`:next &optional` *n* [*Break Loop Command*]
`:n &optional` *n* [*Abbreviated Break Loop Command*]

Makes the *n*-th next visible event the new current event. If there are less than *n* next events, then the last visible event in the invocation sequence becomes the new current event. *n* must be a positive integer and the default is `1`.

`:backtrace` [*Break Loop Command*]
`:b` [*Abbreviated Break Loop Command*]

Prints the event symbols of all visible events in order. The symbol of the current event is printed in upper-case letters and the event symbols of other events are in lower-case.

`:help` [*Break Loop Command*]
`:h` [*Abbreviated Break Loop Command*]

Lists the break loop commands.

`:quit &optional` *n* [*Break Loop Command*]
`:q &optional` *n* [*Abbreviated Break Loop Command*]

Returns control to the level *n* break loop. If *n* is 0 or if *n* is omitted, then control will return to the top-level loop. *n* must be a non-negative integer smaller than the current break level.

`:resume` *[Break Loop Command]*
`:r` *[Abbreviated Break Loop Command]*

Returns control to the caller of the break loop. If the break loop has been entered from `cerror`, `cerror` returns `nil` as its value and control will resume at that point. Otherwise, this command returns control to the previous break loop (or to the top-level loop, if the current break level is 1).

`:variables` *[Break Loop Command]*
`:v` *[Abbreviated Break Loop Command]*

Prints the names of the bound variables in the current environment. To see the value of a bound variable, just type the variable name.

`:functions` *[Break Loop Command]*

Prints the names of the local functions and local macros in the current environment. To see the definition of a local function or macro, use the function special form in the usual way. That is, (`function` *name*) will return the definition of the local function or macro whose name is *name*. Local functions and local macros may be invoked as usual.

`:blocks` *[Break Loop Command]*

Prints the names of the blocks established in the current environment. If a block *block* is established, then the `return-from` form (`return-from` *block value*) works as usual. That is, the block form that established *block* will return *value* as its value and control will resume at that point.

`:tags` *[Break Loop Command]*

Prints the tags established in the current environment. If a tag *tag* is established, then the `go` form (`go` *tag*) works as usual. That is, control will resume at the position of *tag* in the surrounding `tagbody`.

`:local &optional` *n* *[Break Loop Command]*
`:l &optional` *n* *[Abbreviated Break Loop Command]*

If *n* is `0` or if it is omitted, then this command prints the value stored in the main stack entry that is pointed to by the current event environment. *n* is an offset from that entry. If *n* is positive, then the value of the *n*-th next (i.e., toward the top of the main stack) entry is printed. If n is negative, then the value of the — n —-th previous (i.e., toward the bottom of the main stack) entry is printed. n must be an integer. It is an error if the specified entry does not lie between the bottom and the top of the stack.

`:hide` *symbol* [*Break Loop Command*]

Hides all events whose event symbol is *symbol*. In particular, by (`:hide 'lambda`) and (`:hide 'lambda-closure`) , all events become invisible whose event functions are lambda-expressions and closures, respectively. If the event symbol of the current event happens to be *symbol*, then the last previous visible event will become the new current event. *symbol* must be a symbol.

Events of `eval` and `evalhook` may never become invisible and attempts to hide them are simply ignored. It is always the case that the first event function is either `eval` or `evalhook`. Keeping both of them visible is the simplest way to avoid the silly attempts of the user to hide all events.

`:hide-package` *package* [*Break Loop Command*]

Hides all events whose event symbol belongs to the package *package*. *package* may be any object that represents a package, i.e., a package object, a symbol, or a string. If the event symbol of the current event happens to belong to the package *package*, then the last previous visible event will become the new current event. Even if `lisp` package was specified as *package*, events of `eval` and `evalhook` do not become invisible. See the description of `:hide` above.

`:unhide` *symbol* [*Break Loop Command*]

`:unhide` is the inverse command of `:hide`. If, however, *symbol* belongs to one of the `:hide-packaged` packages, events of *symbol* become visible only after the package is `:unhide-package 'd`. *symbol* must be a symbol.

`:unhide-package` *package* [*Break Loop Command*]

`:unhide-package` is the inverse command of `:hide-package`. However, an event whose event symbol belongs to *package* becomes visible only after the symbol is `:unhide 'd`, if the symbol was `:hide 'd` before. *package* may be any object that represents a package, i.e., a package object, a symbol, or a string.

Example:

```
>(defun fact (x) (if (= x 0) one (* x (fact (1- x)))))
fact                    ;;;  Wrong definition for  fact , the factorial.

>(fact 6)               ;;;  Tries to calculate factorial 6.

Error: The variable ONE is unbound.
Error signalled by IF.

Broken at IF:           ;;;  Enters the break-loop.
>>:h                    ;;;  Help.
:c(urrent)              Shows the current function.
:p(revious)             To the previous function.
:n(ext)                 To the next function.
:b(acktrace)            Prints backtrace.
:h(elp)                 Help.
:q(uit)                 Returns to top-level.
:r(esume)               Returns to the caller of break-level.
:l(ocal)                Shows the n-th local value on the stack.
:v(ariables)            Shows local variables.
:functions              Shows local functions.
:blocks                 Shows block names.
:tags                   Shows tags.
:(un)hide(-package)     (Un)hide a function (or a package).

>>:b                    ;;;  Backtrace.
Backtrace: eval > fact > if > fact > if > fact > if > fact >
if > fact > if > fact > if > fact > IF

>>:p                    ;;;  Moves to the previous event.
Broken at FACT.

>>:b                    ;;;  Now inside of  fact  but outside of  if .
Backtrace: eval > fact > if > fact > if > fact > if > fact >
if > fact > if > fact > if > FACT > if

>>:v                    ;;;  Shows local variables.
Local variables: x.

>>x                     ;;;  The value of  x  is  0 .
0

>>:blocks               ;;;  Shows blocks.
Block names: fact.
```

```
>>(return-from fact 1) ;;;  Returns from the  fact  block with value  1 .
720                    ;;;  Now the correct answer.

>                      ;;;  Top-level.
```

5.5 Describe and Inspect

`describe` *object* [*Function*]

Prints the information about *object* to the stream that is the value of `*standard-output*`. The description of an object consists of several fields, each of which is described in a recursive manner. For example, a symbol may have fields such as home package, variable documentation, value, function documentation, function binding, type documentation, `deftype` definition, properties.

`inspect` *object* [*Function*]

Prints the information about *object* in an interactive manner. The output of inspect is similar to that of `describe`, but after printing the label and the value of a field (the value itself is not `describe 'd`), it prompts the user to input a one-character command. The input to `inspect` is taken from the stream that is the value of `*query-io*` . Normally, the inspection of *object* terminates after all of its fields have been inspected. The following commands are supported:

`n` Next. Goes to the next level; the field is inspected recursively.

`s` Skip. Skips the inspection of the field. `inspect` proceeds to the next field.

`p` Print. Pretty-prints the field and prompts again.

`u` *form* Update. The *form* is evaluated and the field is replaced by the resulting value. If the field cannot be updated, the message `Not updated.` will be printed.

`a` Abort. Aborts the inspection of the current object. The field and the rest of the fields are not inspected.

`e` *form* Eval. Evaluates the specified form in the null environment and prints the resulting values. Then prompts again with the same field.

`q` Quit. Aborts the entire inspection.

`?` Help. Lists the `inspect` commands.

Chapter 6

The Compiler

The KCL compiler translates a Lisp program stored in a source file into a C-language program, invokes the C-language compiler to compile the C-language program, and then generates an object file, called *fasl file* (or *o-file* because of the actual filetype). The compiled program in a fasl file is loaded by the function `load`.

Ordinarily, the object program generated by the KCL compiler scarcely does runtime error-checking for runtime efficiency. In addition, Lisp functions in the same source file are linked together and some system functions are open-coded in-line. To control runtime error checking, supply appropriate `optimize` declarations (see Section 7.1).

The KCL compiler processes the `eval-when` special form exactly as specified in the *Common Lisp Reference Manual*. However, all top-level forms in the source file are normally processed in *compile-time-too* mode, not in *not-compile-time* mode (see Section 5.3.3 of the *Common Lisp Reference Manual*). That is, each top-level form *top-level-form* is processed as if it were surrounded by the `eval-when` special form with the situations `compile, load`, and `eval`.

`(eval-when (compile load eval)` *top-level-form*)

There is no exception for this rule. Thus, for instance, in the example of `set-macro-character` form in Section 5.3.3 of the *Common Lisp Reference Manual*, the surrounding `eval-when` form is unnecessary in KCL. If it is desired that each top-level form be processed in *not-compile-time* mode, change the value of the KCL specific variable `*eval-when-compile*` as described below.

The KCL compiler is invoked by the functions `compile-file, compile`, and `disassemble` described below. In addition, the KCL compiler may be invoked directly by the Shell commands `lc` or `lc1`. These commands require the file name of the source file as their argument. Both `lc` and `lc1` simply add "`.lsp`" to the file name argument to obtain the full name of the source file.

% `lc` *filename*

has the same effect as the compiler invocation (`compile-file` "*filename*") from within KCL, and

% `lc1` *filename*

has the same effects as (`compile-file` "*filename*" `:o-file t :c-file t :h-file t :data-file t`).

`compile-file` *input-pathname* [*Function*]

`&key :output-file :output-file :o-file :c-file :h-file :data-file`

`compile-file` compiles the Lisp program stored in the file specified by *input-pathname*, and generates a fasl file. Also `compile-file` generates the following temporary files.

Temporary File	*Contents*
c-file	C version of the Lisp program
h-file	The include file referenced in the c-file
data-file	The Lisp data to be used at load time

If files of these names already exist, the old files will be deleted first. Usually, these intermediate files are automatically deleted after execution of `compile-file`.

The input-file is determined in the usual manner (see Section 2.9), except that, if the filetype is not specified, then the default filetype `.lsp` will be used. The keyword parameter `:output-file` defines the default directory and the default name to be applied to the output files (i.e., the fasl file and the temporary files). `:output-file` itself defaults to *input-pathname*. That is, if `:output-file` is not supplied, then the directory and the name of the input file will be used as the default directory and the default name for the output files. The filetypes of the output files are fixed as follows.

Output File	*Filetype*
fasl file	`.o`
c-file	`.c`
h-file	`.h`
data-file	`.data`

Each output file can be specified by the corresponding keyword parameter. If the value of the keyword parameter is `nil`, then the output file will be deleted after execution of `compile-file`. If the value of the keyword parameter is `t`, then the output file will be left in the default directory under the default name. Otherwise, the output file will be left in the directory under the name specified by the keyword parameter. The default value of `:o-file` is t , and the default values of `:c-file, :h-file`, and `:data-file` are all `nil`.

Example:
(compile-file 'foo)

The source file is "FOO.lsp" and the fasl file is "FOO.o" both in the current directory.

(compile-file 'foo.lish)

The source file is "FOO.LISH" and the fasl file is "FOO.o".

(compile-file "/usr/mas/foo" :output-file "/usr/tai/baa")

The source file is "foo.lsp" in the directory "/usr/mas", and the fasl file is "baa.o" in the directory "/usr/tai".

——— Note to KCL/AOS users ———

The compiler of KCL/AOS generates the following output files.

Output File	*Filetype*	*Contents*
fasl file	`.fasl`	The fasl file
c-file	`.c`	C version of the Lisp program
h-file	`.h`	The include file referenced in the c-file
ob-file	`.ob`	The object file generated by the C compiler
data-file	`.data`	The Lisp data to be used at load time

Note that the filetype of the fasl file is `.fasl` in KCL/AOS. Also note that the compiler of KCL/AOS generates an additional file *ob-file*. Accordingly, the function `compile-file` has a slightly different definition:

`compile-file` *input-pathname* [*Function*]

`&key :output-file :fasl-file :c-file :h-file :ob-file :data-file`

The keyword parameter `:fasl-file` corresponds to the keyword parameter `:o-file` in KCL on Unix. In particular, the default value of `:fasl-file` is t and the default values of `:c-file, :h-file, :ob-file,` and `:data-file` are all `nil`. In KCL/AOS,

```
) lc1  filename
```

has the same effects as (`compile-file` "*filename*" `:fasl-file t :c-file t :h-file t :ob-file t :data-file t`).

Because KCL/AOS follows the convention of the AOS/VS file system, the last example above should be replaced as:

```
( compile-file '':udd:mas:foo'' :output-file '':udd:tai:baa'')
```

The source file is “FOO.LSP” in the directory “:UDD:MAS”, and the fasl file is “BAA.FASL” in the directory “:UDD:TAI”.

———— End of Note ————

`compile` *name* `&optional` *definition* [*Function*]

If *definition* is not supplied, *name* should be the name of a not-yet-compiled function. In this case, compile compiles the function, replaces the previous definition of *name* with the compiled function,*and* returns *name*. If *definition* is supplied, it should be a lambda-expression to be compiled and *name* should be a symbol. If *name* is a non-`nil` symbol, then compile installs the compiled function as the function definition of *name* and returns *name*. If *name* is `nil`, then compile simply returns the compiled function.

The KCL compiler is essentially a file compiler, and forms to be compiled are supposed to be stored in a file. Thus compile actually creates a source file which contains the form designated by the arguments. Then compile calls `compile-file` to get a fasl file, which is then loaded into KCL. The source file and the fasl file are given the names `gazonk.lsp` and `gazonk.fasl`, respectively. These files are not deleted automatically after the execution of `compile:`.

`disassemble &optional` *thing* `&key :h-file :data-file` [*Function*]

This function does not actually disassemble. It always calls the KCL compiler and prints the contents of the c-file, i.e., the C-language code, generated by the KCL compiler. If *thing* is not supplied, or if it is `nil`, then the previously compiled form by disassemble will be compiled again. If *thing* is a symbol other than `nil`, then it must be the name of a not-yet-compiled function, whose definition is to be compiled. In this case, it is an error if the name is associated with a special form or a macro. If *thing* is a lambda-expression (`lambda` *lambda-list . body*), then `disassemble` first creates a function definition (`defun gazonk` *lambda-list . body*) and this definition is compiled. (The function name `gazonk` has no special meanings. Indeed, the displayed code is essentially independent of the function name.) Otherwise, *thing* itself will be compiled as a top-level form. In any case, `disassemble` does not install the compiled function. `disassemble` returns no value.

No intermediate h-file is created if the keyword parameter `:h-file` is `nil` or if `:h-file` is not supplied. Otherwise, an intermediate h-file is created under the name specified by `:h-file`. Similarly, the intermediate data-file is specified by the keyword parameter `:data-file`.

`*eval-when-compile*` [*Variable*]

The compiler processes each top-level form in *not-compile-time* mode if the value of this variable is `nil`, and in *compile-time-too* mode, otherwise. See Section 5.3.3 of *Common Lisp Reference Manual* for these two modes. The initial value of this variable is `t`.

Chapter 7

Declarations

KCL supports all kinds of declarations described in the *Common Lisp Reference Manual*. Any valid declaration will affect the KCL environment in some way or another, although information obtained by declarations, other than special declarations, is mainly used by the KCL compiler.

As described in the *Common Lisp Reference Manual*, Common Lisp declarations are divided into two classes: *proclamations* and others. A proclamation is a global declaration given by the function `proclaim`, the top-level *macro* `defvar`, or the top-level macro `defparameter`. Once given, a proclamation remains effective during the KCL session unless it is shadowed by a local declaration or is canceled by another proclamation. Any other declaration is a *local declaration* and is given only by the special form `declare`. A local declaration remains in effect only within the body of the construct that surrounds the declaration. In the following nonsensical example borrowed from Chapter 9 of the *Common Lisp Reference Manual*,

```
(defun nonsense (k x z)
   (foo z x)
   (let ((j (foo k x))
         (x (* k k)))
        (declare (inline foo) (special x z))
     (foo x j z)))
```

the `inline` and the special declarations both remain in effect within the surrounding `let` form. In this case, we say that the `let` form is the *surrounding construct* of these declarations.

`proclamation` *decl-spec* [*Function*]

This function is introduced to KCL so that the user can see currently effective proclamations. The argument *decl-spec* specifies the proclamation to be checked. It may be any declaration specification that can be a valid argument to the function **proclaim**. The function **proclamation** returns t if the specified proclamation is still in effect. Otherwise, it returns **nil**. For example,

```
>(proclaim '(special *x*))  ;;;  The variable  *x*  is
nil                         ;;;  proclaimed to be globally special.

>(proclamation '(special *x*))
t

>(defvar *y*)                ;;;  Another way to proclaim a variable
nil                          ;;;  to be globally special.

>(proclamation '(special *y*))
t
```

`the` *value-type form* [*Special Form*]

The KCL interpreter does actually check whether the value of the *form* conforms to the data type specified by *value-type* and signals an error if the value does not. The type checking is performed by the function `typep`. For example,

```
   (the fixnum (foo))

  is equivalent to

  (let ((values (multiple-value-list (foo))))
     (cond ((endp values) (error ``Too few return values.''))
           ((not (endp (cdr values)))
            (error ``Too many return values.''))
           ((typep (car values) 'fixnum) (car values))
           (t (error ``~s is not of type fixnum.'' (car values)))))
```

On the other hand, the KCL compiler uses the the special form to obtain type information for compiled code optimization. No code for runtime type-checking is embedded in the compiled code.

7.1 Declaration Specifiers

KCL recognizes all declaration specifiers defined in the *Common Lisp Reference Manual.* The syntax of each such declaration specifier is exactly the same as defined in the *Common Lisp Reference Manual*. In addition, KCL recognizes the `object` declaration specifier which is specific to KCL.

`special` { *variable-name* }* [*Declaration Specifier*]

The interpreter and the compiler of KCL both treat special declarations exactly as described in the *Common Lisp Reference Manual*.

`type` *type* { *variable-name* }* [*Declaration Specifier*]

A `type` proclamation (`type` *type var1 var2* ...) specifies that the dynamic values of the named variables are of the type *type*. A local `type` declaration specifies that the variables mentioned are bound by the surrounding construct and have values of the type *type* during execution of the surrounding construct. The compiler issues a warning if one of the named variables is not bound by the surrounding construct. The information given by `type` declarations is used by the compiler to optimize the compiled code. The behavior of the compiled code is unpredictable if a wrong `type` declaration is supplied. The compiler detects certain wrong `type` declarations at compile time. For example,

```
>(defun foo (x y)
   (declare (fixnum x) (character y))
   (setq x y)
  ...))
foo

>(compile 'foo)

; (defun foo ...) is being compiled.
;; Warning: Type mismatches between x and y.}
```

See Section 7.3 for further information on `type` declarations.

type {*variable-name* }* [*Declaration Specifier*]

(*type var1 var2* ...) is equivalent to (`type` *type var1 var2* ...), provided that *type* is one of the symbols in Table 4-1 of the *Common Lisp Reference Manual*, other than `function`. Declaration specifications that begin with `function` are regarded as `function` declarations (see below).

`function` *function-name argument-types. return-types* [*Declaration Specifier*]

A `function` declaration is used to obtain type information for function call forms. That is, a `function` declaration specifies the argument and the return types of each form that calls the named function.

```
(defun foo ()
  (declare (function bar (character) fixnum))
  (+ (bar (atcholi1)) (bar (atcholi2))))
```

In this example, the `function` declaration specifies that the two functions `atcholi1` and `atcholi2` both return character objects when called within the body of `foo`, and that the function bar returns fixnum objects when called within the body of `foo`. The type information given by function declarations is used by the compiler to optimize the compiled code. The behavior of the compiled code is unpredictable if a wrong `function` declaration is supplied. The compiler detects certain wrong `function` declarations at compile time. For example,

```
>(defun foo (x)
   (declare (fixnum x)
            (function bar (character) fixnum))
   (bar x))
foo

>(compile 'foo)

; (defun foo ...) is being compiled.
;; Warning: The type of the form x is not character.
```

However, the compiler does not check the number of arguments, and thus, the following function definition will be compiled successfully without any warnings.

```
(defun foo ()
  (declare (function bar (character character) fixnum))
  (+ (bar (atcholi1)) (bar (atcholi2) (atcholi3) (atcholi4))))
```

For this definition, the compiler assumes that the three functions `atcholi1`, `atcholi2`, and `atcholi3` will return fixnum objects. The return type of `atcholi4` is unknown at compile time.

The complete syntax of a function declaration is:

```
(function  function-name
  ( { type }* [ { &optional | &rest | &key } { thing }* ] )
  { (values { type }* ) | { type }* }
)
```

Although **&optional**, **&rest**, and **&key** markers may appear in the list of argument types, only those *types* are recognized that appear before any such markers and the rest of the list is simply ignored. Note that functions with **&optional**, **&rest**, or **&key** parameters may still be declared by **function** declarations because of the use of **function** declarations mentioned above.

The **values** construct in the specification of return types is almost useless: (**function** *function-name argument-types* (**values** *type1 type2* ...)) is equivalent to (**function** *function-name argment-types type1 type2* ...) . We, the implementors of KCL wonder why the value construct was introduced in Common Lisp.

See Section 7.3 for further information on **function** declarations.

ftype *function-type* {*function-name*}* [*Declaration Specifier*]

function-type must be a list whose first element is the symbol **function**. (**ftype (function**. *rest) function-name-1 ... function-name-n*) is equivalent to *n* consecutive **function** declarations (**function** *function-name-1 . rest*) ... (**function** *function-name-n . rest*).

notinline {*function-name*}* [*Declaration Specifier*]

(**notinline** *function1 function2* ...) specifies that the compiler should not compile the named functions in-line. Calls to the named functions can be traced and an event (see Section 5.4) is pushed on the event stack when any one of the named functions is invoked.

inline {*function-name*}* [*Declaration Specifier*]

An **inline** proclamation cancels currently effective **notinline** proclamations, and a local **inline** declaration locally shadows currently effective **notinline** declarations.

```
>(defun foo (x)
   (cons (car x)
         (locally (declare (inline car)) (car x))))
foo

>(defun bar (x)
   (cons (car x)
         (locally (declare (inline car)) (car x))))
foo
```

```
>(proclaim '(notinline car))
nil

>(compile 'foo)
...

>(proclaim '(inline car))
nil

>(compile 'bar)
...
```

Usually, primitive functions such as car are compiled in-line. Therefore, in this example, only the first call to `car` within `foo` is compiled not in-line.

In general, the KCL compiler compiles functions in-line whenever possible. Thus an `inline` declaration (`inline` *function1 function2* ...) is worthless if none of the named functions have previously been declared to be `notinline` .

`ignore` {*variable-name*}* [*Declaration Specifier*]

Usually, the compiler issues a warning if a lexical variable is never referred to. (`ignore` *var1* ... *varn*) causes the compiler not to issue a warning even if the named variables are never referred to. The compiler issues a warning if one of the named variables is not bound by the surrounding construct, or if a named variable is actually referred to. `ignore` proclamations are simply ignored.

`optimize` { { (*quality value*) | *quality*}}* [*Declaration Specifier*]

KCL supports the four `optimize` qualities listed in the *Common Lisp Reference Manual*. `speed` and `compilation-speed` are used to set up the optimization switch of the C language compiler which is invoked to compile the C-language code generated by the KCL compiler (see Chapter 6). (`optimize (speed` *n*)) and (`optimize (compilation-speed` it m)) are equivalent, where *n* and *m* are integers between 0 and 3 , and *m* is equal to 3-*n*. When a KCL session is started, the `speed` quality is set to 3. That is, by default, the compiler generates the fastest code in the longest compilation time. The `space` quality specifies whether the code size is important or not: The compiled code is a little bit larger and faster when compiled with the space quality 0, than when compiled with the space quality 1, 2, or 3. When a KCL session is started, the `space` quality is set to 0. The `safety` quality determines how much runtime error checking code should be embedded in the compiled code. If the `safety` quality is 0, the compiled code scarcely does runtime error checking. If the `safety` quality is 1 , then the compiled code for a function will check the number of arguments to the function at runtime. If the `safety` quality is 2 or 3, then the compiled code does full runtime error checking. In addition, the highest quality value 3 causes the compiler to treat all functions as if they were declared to be **`notinline`**. When a KCL session is started, the `safety` quality is set to 0.

`declaration` {*name*}* [*Declaration Specifier*]

A `declaration` declaration is used exactly as specified in the *Common Lisp Reference Manual.*

`object` {*variable-name*}* [*Declaration Specifier*]

This is the only declaration specifier that is specific to KCL. (`object` *var1 ... varn*) affects only variable bindings and specifies that the named variables can be allocated in the C stack (see Section 7.3). The compiler issues a warning if one of the named variables is not bound by the surrounding construct. `object` proclamations are simply ignored.

7.2 Significant Type Specifiers

Whenever a declaration is encountered, each type specifier (if any) in the declaration is converted to one of the following type specifiers, which are collectively called the *significant type specifiers.*

```
       |------------   fixnum
       |------------   character
       |------------   short-float
       |------------   long-float
   t --|----   (array t)  --------------  (vector t)
       |----   (array fixnum)  ---------  (vector fixnum)
       |----   (array string-char)   ---  string
       |----   (array short-float)   ---  (vector short-float)
       |----   (array long-float)    ---  (vector long-float)
       |----   (array bit)   -----------   bit-vector
```

Here, the lines indicate subtype relations; the right type is a subtype of the left type. For instance, (`vector t`) is a subtype of (`array t`) and `t`, and (`array t`) itself is a subtype of `t`. However, (`array t`) and (`array string-char`) are disjoint types.

The function `subtypep` is used for the conversion to significant type specifiers: If the first value of (`subtypep` *raw-type type*) is `t` for one of the significant type specifiers *type*, then the type specifier *raw-type* in the declaration is converted to *type*. If there are more than one such significant type specifiers, then the type specifier that is a subtype of other specifiers is selected. For example, type specifiers `fixnum, (mod 3)`, and (`member 0 1`) are all converted to `fixnum`, though they are also subtypes of `t`.

Because of this type specifier conversion, KCL may sometimes regard two seemingly distinct declarations as the same. For example, the following `type` declarations are completely equivalent, internally in KCL.

```
(declare (type fixnum x))

(declare (type (mod 3) x))

(declare (type (member 0 1) x))
```

Type specifiers in declaration specifications passed to the KCL specific function `proclamation` are also converted to significant type specifiers. Thus, for example,

```
>(proclaim '(function foo (fixnum) fixnum))
nil

>(proclamation '(function foo ((mod 3)) (member 0 1)))
t

>(proclamation '(function foo (number) character))
nil
```

The first call to `proclamation` returns `t` because both (`mod 3`) and (`member 0 1`) are converted to `fixnum` before the function type of `foo` is checked.

7.3 Treatment of Type Declarations

KCL has several runtime stacks. One of them is called the *value stack* which is the "main stack" of KCL: Arguments to functions and resulting values of functions are usually passed via the value stack, lexical variables in compiled code are usually allocated on the value stack, and temporary values during evaluation of nested expressions are usually saved on the value stack. However, if appropriate declarations are supplied to the compiler, the compiled code will use another stack called the *C stack*, which can be accessed more efficiently than the value stack. In addition, arguments and resulting values passed via the C stack, values of lexical variables allocated on the C stack, and temporary values saved on the C stack may sometimes be represented as *raw data* instead of pointers to heap-allocated cells. In KCL, even a fixnum object is usually represented as a pointer to a fixnum cell in which the raw datum (i.e., the 32-bit signed integer) for the fixnum is stored. Accessing such raw data on the C stack results in faster compiled code, partly because no pointer deferencing operation is necessary, and partly because no cell is newly allocated on the heap when a new object is created. In contrast, any object on the value stack is represented as a pointer to a heap-allocated cell.

One of the deficiencies of the use of the C stack is that raw data on the C stack may sometimes need to be reallocated on the heap. Suppose, in the following example, that the lexical variable `x` is allocated on the C stack and has always a fixnum raw datum as its value. (The situations in which this occurs will be explained later.)

```
(defun foo ()
  (let ((x 0))
    ....
    (bar x)
    ....
    ))
```

Also suppose that the function bar expects its argument to be passed via the value stack rather than via the C stack. (This situation typically occurs when `foo` and `bar` are defined in separate source files. See below.) On call to `bar`, the compiled code of `foo` will allocate a fixnum cell on the heap and push the pointer to this cell on the value stack as the argument to `bar`.

Another deficiency is that it is sometimes dangerous to allocate a cell pointer onto the C stack. (This occurs when `object` declarations are supplied. See below.) The garbage collector of KCL never takes care of cell pointers on the C stack and thus a heap-allocated cell pointed to only from the C stack may be recycled for further use, while the data in the cell is still in use. This is why KCL usually uses the less efficient value stack. In contrast, objects on the value stack are automatically protected against garbage collection. Note that raw data on the C stack need not be protected against garbage collection because they remain alive until the C stack is popped.

7.4 Variable Allocations

If a lexical variable is declared to be of `fixnum` , `character` , `short-float` , `long-float`, or their subtypes, then it is allocated on the C stack rather than on the value stack. In addition, the variable always has a raw datum as its value: 32 bit signed integer for fixnums, 8 bit character code with 24 bit padding for characters (remember that the font and bit fields of KCL characters are always 0

), 32 bit floating point representation for short-floats, and 64 bit floating point representation for long-floats. Similarly, if a lexical variable is named in an `object` declaration (see Section 7.1), then it is allocated on the C stack but, in this case, the variable always has a cell pointer as its value. The user is strongly recommended to make sure that objects stored in such an `object` variable may never be garbage collected unexpectedly. For example,

```
(do ((x (foo) (cdr x)))
    ((endp x))
    (let ((y (car x)))
         (declare (object y))
      (bar y)))
```

this `object` declaration is completely safe because the value of the variable `y` is always a substructure of the value of `x`, which in turn is protected against garbage collection. Incidentally, loop variables of `dolist` may always be declared as object variables, since the `dolist` form has essentially the same control structure as the `do` form above. On the other hand, the result of evaluation of the following form is unpredictable, because the cons cell pointed to from the `object` variable `z` may be garbage collected before `bar` is called.

```
(let ((z (cons x y)))
     (declare (object z))
  (foo (cons x y))
  (bar z))
```

Lexical variables that are not declared to be of `fixnum, character, short-float, long-float`, or their subtypes, and that are not named in `object` declarations are usually allocated on the value stack, but may possibly be allocated on the C stack automatically by the compiler.

7.5 Built-in Functions that Operate on Raw Data Directly

Some built-in Common Lisp functions can directly operate on raw data, if appropriate declarations are supplied. The addition function + is among such functions.

```
(let ((x 1))
     (declare (fixnum x))
  ....
  (setq x (+ x 2))
  ....
  )
```

In the compiled code for this `let` form, the raw fixnum datum (i.e., the 32 bit signed integer) stored in `x` is simply incremented by 2 and the resulting 32 bit signed integer is stored back into `x`. The compiler is sure that the addition for 32 bit signed integers will be performed on the call to +, because the arguments are both fixnums and the return value must be also a fixnum since the value is to be assigned to the `fixnum` variable. The knowledge of both the argument types and the return type is necessary for this decision: Addition of two fixnums may possibly produce a bignum and addition of two bignums may happen to produce a fixnum value. If either the argument type or the return type were not known to the compiler, the general addition function would be called to handle the general case. In the following form, for example, the compiler cannot be sure that the return value of the multiplication is a fixnum or that the arguments of the addition are fixnums.

```
(setq x (+ (* x 3) 2))
```

In order to obtain the optimal code, a `the` special form should surround the multiplication.

```
(setq x (+ (the fixnum (* x 3)) 2))
```

Built-in Common Lisp functions that can directly operate on raw data are:

1. arithmetic functions such as `+, -, 1+, 1-, *, floor, mod`, /, and `expt`.
2. predicates such as `eq, eql, equal, zerop, plusp, minusp, =, /=, <, <=, >, >=, char=, char/=, char<, char<=, char>`, and `char>=`.
3. sequence processing functions that receive or return one or more fixnum values, such as `nth, nthcdr, length`, and `elt`.
4. array access functions such as `svref, char, schar`, and `aref` (see below).
5. system-internal functions for array update (see below).
6. type-specific functions such as `char-code, code-char`, and `float`.

As mentioned in Section 2.5.1, array elements are represented in one of six ways depending on the type of the array. By supplying appropriate array type declarations, array access and update operations can handle raw data stored in arrays. For example,

```
(let ((a (make-array n :element-type 'fixnum))
      (sum 0))
     (declare (type (array fixnum) a)
              (fixnum sum))
  (dotimes (i n)              ;;; Array initialization.
```

```
            (declare (fixnum i))
      (setf (aref a i) i))
   ....
   (dotimes (i n)              ;;; Summing up the elements.
            (declare (fixnum i))
      (setq sum (+ (aref a i) sum)))
   ....
   )
```

The `setf` form replaces the `i-th` element of the array a by the raw fixnum value of `i`. The `aref` form retrieves the raw fixnum datum stored in `a`. This raw datum is then added to the raw fixnum value of the fixnum variable `sum`, producing the raw fixnum datum to be stored in `sum`. Similar raw data handling is possible for arrays of types `(array fixnum)` , `(vector fixnum)` , `(array string-char)` , `string` , `(array short-float)` , `(vector short-float)` , `(array long-float)`, and `(vector long-float)` .

7.6 Arguments/Values Passing

Function proclamations (`function` *function-name (arg-type1 arg-type2 ...) return-type*) or its equivalents give the compiler the chance to generate compiled code so that arguments to the named functions and resulting values of the named functions will be passed via the C stack, thus increasing the efficiency of calls to these functions. Such arguments/values passing via the C stack is possible only if the called function is also defined in the same source file. This is because the code for the called function must have two entries: One entry for arguments/values passing via the C stack and another for ordinary arguments/values passing via the value stack. (An ordinary function has only the latter entry.) When the latter entry is used, the arguments on the value stack are pushed onto the C stack and then the former entry is used to execute the body of the function. On return from the function, the resulting value on the C stack is pushed onto the value stack. This means that ordinary calls to these functions are slower than calls to ordinary functions.

One of the merits of arguments/values passing via the C stack is that raw data stored in C-stack-allocated variables can be passed directly to other functions and raw data returned from functions may directly be saved in C-stack-allocated variables or may directly be used as arguments to another function. A good example of this follows.

```
(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))))

;;; Call (tak 18 12 6).
```

When `tak` is called with the arguments `18, 12`, and `6`, the raw fixnum data of the arguments are set to the parameters `x, y, z` which are allocated on the C stack. After that, only raw data on the C stack are used to perform the execution: No cell pointers are newly allocated nor even referenced. Arguments and resulting values for recursive calls to `tak` are passed via the C stack, and the built-in functions < and `1-` directly operate on the raw data. Only at the return from the top-level call of `tak`, the resulting raw data value (which happens to be `7`) is reallocated on the heap. Note that both the `function` proclamation and the local `fixnum` declaration are necessary to obtain the optimal code. The `function` proclamation is necessary for arguments/values passing via the C stack and the `fixnum` declaration is necessary to allocate the parameters onto the C stack.

Chapter 8

Operating System Interface

KCL provides the following facilities that are not defined in the *Common Lisp Reference Manual.*

`save` *filename* [*Function*]

`save` saves the current memory image into a program file *filename*. After saving the memory image, the KCL process terminates immediately. To execute the saved program file, specify the full pathname of the file, as indicated in the example below.

——— Note to KCL/AOS users ———

In KCL/AOS, if the filename does not contain the filetype `.pr`, then the program file is given the name *filename* `.pr`. Also `save` saves a symbol table file with the filetype `.st`. When the program file is executed, execution begins at the top-level of KCL. Even if there were streams that were open at the call of `save`, these streams are not effective when the program file is executed. The function `save` of KCL/AOS does not terminate the KCL process.

——— End of Note ———

Example:

```
>(defun plus (x y) (+ x y))
plus

>(save ``savefile'')
%

% pwd
```

```
/usr/hagiya
% /usr/hagiya/savefile

>(plus 2 3)
5

>(bye)
Bye.
%
```

———— Note to KCL/AOS users ————

Here is the example of save in KCL/AOS.

```
>(defun plus (x y) (+ x y))
plus

>(save ``savefile'')
t

>(bye)
Bye.
)

) X SAVEFILE

>(plus 2 3)
5

>(bye)
Bye.
)
```

———— End of Note ————

`system` *string* [*Function*]

Executes a Shell command as if *string* is an input to the Shell. On return from the Shell command, `system` returns the exit code of the command as an integer.

———— Note to KCL/AOS users ————

This function is not supported in KCL/AOS. Use instead those functions specific to KCL/AOS described below.

———— End of Note ————

`bye &optional` *exit-code* [*Function*]
`by &optional` *exit-code* [*Function*]

Terminates KCL and returns the *exit-code* to the parent process. *exit-code* must be an integer and its default value is 0 .

———— Note to KCL/AOS users ————

The functions `bye` and `by` of KCL/AOS accept a string instead of an *exit-code*. If *string* is supplied, these functions return the *string* to the father as the termination message.

———— End of Note ————

The following functions `process, termination-message`, and `last-termination-message` are specific to KCL/AOS and are not supported in KCL on Unix.

`process` *progname* `&optional` *ipc-message* [*Function*]

```
\&key    :block :console :debug :dir
         :input :output :username
         :list :data :ioc
```

This function is defined only in KCL/AOS. `process` creates a process in the way specified by its arguments.

progname The name of the process to create. Must be a string.

ipc-message The IPC message passed to the process. Must be a string. Arguments in the message must be separated with commas ' , '.

:block If `non- nil`, KCL blocks its execution while the son executes. Defaults to `t`.

:console The name of the file to be associated to @`console`. Must be a string. No file is associated if `:console` is not specified and if the `:ioc` argument (see below) is specified with a non-`nil` value. `:console` supercedes `:ioc`.

:debug If non-`nil`, the son runs in the debug mode.

:input The name of the file to be associated to @`input`. Must be a string. No file is associated if `:input` is not specified and if the `:ioc` argument (see below) is specified with a non-`nil` value. `:input` superceds `:ioc`.

:output The name of the file to be associated to @`output`. Must be a string. No file is associated if `:output` is not specified and if the `:ioc` argument (see below) is specified with a non-`nil` value. `:output` supercedes `:ioc`.

:username The user-name of sub-process. Must be a string. If not specified, the current user-name is used.

:list If a string, the name of the file to be associated to @`list`. If `t`, the file currently associated to @`list` is used. If `nil`, or if not specified, no list file is passed to the son.

:data If a string, the name of the file to be associated to @`data`. If `t`, the file currently associated to @`data` is used. If `nil`, or if not specified, no data file is passed to the son.

:ioc If non-`nil, current` @`input`, @`output`, and @`console` files are passed to the son. If not specified, these files are not passed unless specified by the `:console, :input`, or `:output` arguments.

Example:

```
>(process ``:cli.pr'')

AOS/VS CLI  REV 03f.03.00.00   19-APRIL-84   12:00:00
)by
AOS/VS CLI  TERMINATING        19-APRIL-84   12:00:00
t

>
```

`termination-message` [*Function*]

This function is defined only in KCL/AOS. `termination-message` returns a string consisting of the termination message of a son. Used in connection with the function `process`.

`last-termination-message` [*Function*]

This function is defined only in KCL/AOS. `last-termination-message` flushes all the messages currently spooled and returns the last termination message.

Chapter 9

Macros

9.1 System Macros

The KCL interpreter implements the following system macros as if they were special forms. That is, macro forms of the following macros are directly evaluated without being macro-expanded.

```
and        case       cond       decf       defmacro   defun
do         do*        dolist     dotimes    incf       locally
loop       multiple-value-bind              multiple-value-list
multiple-value-setq              or         pop        prog
prog*      prog1      prog2      psetq      push       return
setf       unless     when
```

For these macro forms, the functions `macro-function` and `special-form-p` both return non-`nil` values: `macro-function` returns the macro expansion function and `special-form-p` returns `t`. Of course, functions such as `macroexpand` and `macroexpand-1` will successfully expand macro forms for these system macros.

9.2 Defmacro Lambda-Lists

A *defmacro lambda-list* is a lambda-list-like construct that is used as the third element in the `defmacro` form,

(`defmacro` *name defmacro-lambda-list* {*declaration* | *doc-string*}* { *form* }*)

The description of defmacro lambda-lists in the *Common Lisp Reference Manual* is quite ambiguous. KCL employs the following syntax.
The complete syntax of a defmacro lambda-list is:

```
( [ &whole var ]
  [ &environment var ]
  { pseudo-var }*
  [ &optional { var | ( pseudo-var [ initform [ pseudo-var ] ] ) }* ]
  { [ { &rest | &body } pseudo-var ]
    [ &key { var | ( { var | ( keyword pseudo-var ) }
                     [ initform [ pseudo-var ] ] ) }*
     [ &allow-other-keys ] ]
      [ &aux { var | ( pseudo-var [ initform ] ) }* ]
 | . var }
)
```

where *pseudo-var* is either a symbol or a list of the following form:

```
( { pseudo-var }*
  [ &optional { var | ( pseudo-var [ initform [ pseudo-var ] ] ) }* ]
  { [ { &rest | &body } pseudo-var ]
    [ &key { var | ( { var | ( keyword pseudo-var ) }
                     [ initform [ pseudo-var ] ] ) }*
     [ &allow-other-keys ] ]
      [ &aux { var | ( pseudo-var [ initform ] ) }* ]
  | . var }
)
```

The defmacro lambda-list keyword `&whole` may appear only at the top-level, first in the defmacro lambda-list. It is not allowed within *pseudo-var*. Use of the `&whole` keyword does not affect the processing of the rest of the defmacro lambda-list:

```
(defmacro foo (&whole w x y) ... )
```

and

```
(defmacro foo (x y) ... )
```

both bind the variables `x` and `y` to the second and the third elements, respectively, of macro forms of `foo`.
The defmacro lambda-list keyword `&environment` may appear only at the top-level, first in the defmacro lambda-list if `&whole` is not supplied, or immediately after the variable that follows `&whole`, if `&whole` is supplied. `&environment` is not allowed within *pseudo-var*. Like `&whole`, use of `&environment` does not affect the processing of the rest of the defmacro lambda-list. If an `&environment` parameter is supplied andif this parameter is not used at all, then the KCL compiler will issue a warning. To suppress the warning, just remove the parameter from the defmacro lambda-list, or add an `ignore` declaration.

The defmacro lambda-list keyword `&body` is completely equivalent to the `&rest` keyword. KCL takes no special action for `&body` parameters.

Although useless, KCL allows supplied-p parameters to be destructured. This is useless because supplied-p parameters can never be bound to a non-empty list. Our intention is to stick to the specification in the *Common Lisp Reference Manual* as far as possible, even if it is silly to do so.

Like for ordinary lambda-lists, the interpreter detects invalid arguments to macro expansion functions. When a parameter is destructured, the structure of the corresponding argument is also checked. Such runtime argument checking may or may not be embedded in compiled code, depending on the environment when the code was generated. If the code was generated while the `safety` optimize level is zero (that is, while the value of (`proclamation '(optimize (safety 0))) is t`), then the generated code does not perform argument checking at all. Otherwise, the compiled code does check the validity of arguments.

Chapter 10

The C Language Interface

This chapter describes the facility of KCL to interface the C language and KCL. With this facility, the user can arrange his or her C-language programs so that they can be invoked from KCL. In addition, the user can write Lisp function definitions in the C language to increase runtime efficiency.

The basic idea of interfacing the C language is this: As mentioned in Chapter 6, the KCL compiler, given a Lisp source file, creates an intermediate C-language program file, called *c-file*, which is then compiled by the C-language compiler to obtain the final fasl-file. Usually, the c-file consists of C-language function definitions. The first C-language function in the c-file is the "initializer", which is executed when the fasl file is loaded, and the other C-language functions are the C versions of the Lisp functions (including macro expansion functions) defined in the source file. By using the top-level macros `Clines` and `defCfun` described below, the user can direct the compiler to insert his or her own C-language function definitions and/or C-language preprocessor macros such as `#define` and `#include` into the c-file. In order that such C-language functions be invoked from KCL, another top-level macro `defentry` is used. This macro defines a Lisp function whose body consists of the calling sequence to the specified C-language function.

The C-language function definitions are placed in the c-file in the order of the corresponding Lisp functions defined in the source file. That is, the C code for the first Lisp function comes first, the C code for the second Lisp function comes second, and so on. If a `Clines` or `defCfun` macro form appears between two Lisp function definitions in the source file, then the C code specified by the macro is placed in between the C code for the Lisp functions.

We define some terminology here which is used throughout this Chapter. A *C-id* is either a Lisp string consisting of a valid C-language identifier, or a Lisp symbol whose print-name, with all its alphabetic characters turned into lower case, is a valid C identifier. Thus the symbol `foo` is equivalent to the string `''foo''` when used as a C-id. Similarly, a *C-expr* is a string or a symbol that may be regarded as a C-language expression. A *C-type* is one of the Lisp symbols `int`, `char`, `float`, `double`, and `object`. Each corresponds to a data type in the C language; `object` is the type of Lisp object and other C-types are primitive data types in the C language.

`Clines` {*string*}* [*Macro*]

When the KCL compiler encounters a macro form (`Clines` *string1 ... stringn*), it simply outputs the *strings* into the c-file. The arguments are not evaluated and each argument must be a string. Each *string* may consist of any number of lines, and separate lines in the *string* are placed in separate lines in the c-file. In addition, each *string* opens a fresh line in the c-file, i.e., the first character in the *string* is placed at the first column of a line. Therefore, C-language preprocessor commands such as `#define` and `#include` will be recognized as such by the C compiler, if the ' # ' sign appears as the first character of the *string* or as the first character of a line within the *string*.

In order to clearly distinguish C code from other parts of Lisp programs, we, the implementors of KCL, make it our rule to start each C code line with a percent sign ' % '. We define % as a read macro which returns the rest of the line as a string. For example,

```
;;; C version of TAK.
(Clines

%       int tak(x, y, z)
%       int x, y, z;
%       {       if (y >= x) return(z);
%               else return(tak(tak(x-1, y, z),
%                               tak(y-1, z, x),
%                               tak(z-1, x, y)));
%       }
)
```

Of course, the user may instead enclose each C code line or the whole C code with double quotes, but we recommend the use of the percent sign read macro. Since the percent sign read macro is not a standard read macro, the users must define this read macro by themselves. We use the following definition.

```
(set-macro-character
    #\\%
    #'(lambda (stream char) (values (read-line stream)))))
```

Here, the lambda-expression returns the first value of `read-line` by using `values` as a filter.

When interpreted, a `Clines` macro form expands to `nil`.

`defentry` *function parameter-list C-function* [*Macro*]

`defentry` defines a Lisp function whose body consists of the calling sequence to a C-language function. *function* is the name of the Lisp function to be defined, and *C-function* specifies the C function to be invoked. *C-function* must be either a list (*type C-id*) or *C-id*, where *type* and *C-id* are the type and the name of the C function. *type* must be a C-type or the symbol `void` which means that the C function returns no value. (`object` *C-id*) may be abbreviated as *C-id*. *parameter-list* is a list of C-types for the parameters of the C function. For example, the following `defentry` form defines a Lisp function `tak` from which the C function `tak` above is called.

```
(defentry tak (int int int) (int tak))
```

The Lisp function `tak` defined by this `defentry` form requires three arguments. The arguments are converted to `int` values before they are passed to the C function. On return from the C function, the returned `int` value is converted to a Lisp integer (actually a fixnum) and this fixnum will be returned as the value of the Lisp function. See below for type conversion between Lisp and the C language.

A `defentry` form is treated in the above way only when it appears as a top-level form of a Lisp source file. Otherwise, a `defentry` form expands to `nil`.

`defla` *name lambda-list* {*declaration* | *doc-string*}* {*form*}* [*Macro*]

When interpreted, `defla` is exactly the same as `defun`. That is, (`defla` *name lambda-list . body*) expands to (`defun` *name lambda-list . body*) . However, `defla` forms are completely ignored by the compiler; no C-language code will be generated for `defla` forms. The primary use of `defla` is to define a Lisp function in two ways within a single Lisp source file; one in the C language and the other in Lisp. `defla` is short for *DEF*ine *L*isp *A*lternative.

Suppose you have a Lisp source file whose contents are:

```
;;; C version of TAK.
(Clines
```

```
%       int tak(x, y, z)
%       int x, y, z;
%       {       if (y >= x) return(z);
%               else return(tak(tak(x-1, y, z),
%                               tak(y-1, z, x),
%                               tak(z-1, x, y)));
%       }

)

;;;  TAK calls the C function tak defined above.
(defentry tak (int int int) (int tak))
;;;  The alternative Lisp definition of TAK.
(defla tak (x y z)
   (if (>= y x)
       z
       (tak (tak (1- x) y z)
            (tak (1- y) z x)
            (tak (1- z) x y))))
```

When this file is loaded into KCL, the interpreter uses the Lisp version of the `tak` definition. Once this file has been compiled, and when the generated fasl file is loaded into KCL, a function call to tak is actually the call to the C version of `tak`.

`defCfun` *header n* {*element*}* [*Macro*]

`defCfun` defines a C-language function which calls Lisp functions and/or which handles Lisp objects. *header* is a string consisting of the C code for

the optional *type-specifier* of the C function,

the *function-declarator* of the C function, and

the *type-decl-list* of the parameters to the C function.

(For the C-language terminology, refer to *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.) The rest of the C function definition, i.e., the *function-statement*, is given by *elements*. Each *element* may be a string, in which case the string is treated in the same way as the arguments to the `Clines` macro. Or else, the *element* is a list ((*name arg1 ... argn*) *place1 ... placem*) . The compiler translates this list into a calling sequence to the Lisp function whose name is *name*. As will be mentioned later, *name* may be `quote`, but *name* may not be the name of any other special form or a macro. The *args* specify the arguments to the function and the *places* specify where the values should go. Thus the list-formed *element* could be regarded as something like the Lisp form:

```
(multiple-value-setq
    ( place1  ...  placem )
    ( name   arg1  ...  argn )).
```

Each *arg* is a list (*C-type C-expr*), where *C-expr* is any C-language expression of the type *C-type*. If *type* is `object`, then *arg* may be written simply as *C-expr*. Similarly, each *place* is a list (*C-type C-expr*), or it may be abbreviated as *C-expr* if *C-type* is `object`. The *C-expr* in this case is any *lvalue* (in the terminology of the C language), i.e., it may be any valid C-language code that can be written at the left side of an assignment.

The function call is performed as follows. The *args* are evaluated, and the values are sent to the specified Lisp function after type conversion from C to Lisp. On return from the called Lisp function, each returned value is assigned to the corresponding *place*, i.e., the first returned value goes to *place1*, the second to *place2*, and so on. If there are more *places* than the values returned, extra values of `nil` are assigned to the remaining *places*. If there are more values than *places*, the excess values are simply discarded. If necessary, Lisp-to-C type conversion may take place before each returned value is assigned.

If the Lisp function is called just for side-effects, then the list-formed *element* may be abbreviated as a one-level list (*name arg1 ... argn*).

As a special case, if a list-formed *element* is of the form ((`quote` *value) place*), the Lisp object *value* is assigned to *place*. Here *value* may be any Lisp object.

The following `defCfun` form defines the C function silly which adds 100 to the value of the parameter `x` and prints the result in three different ways. The second argument to `defCfun` will be described later, and the user may ignore it.

```
(defCfun ''silly(x) int x;'' 0
%       int y;
        ((+ (int x) (int ''100'')) (int y))
%       printf(''\\n%d'', y);
%       y = x+100;
        (print (int y))
        (print (int ''x+100''))
 )
```

When a C function handles Lisp objects (i.e., data of type `object`), the user should be careful enough so that the objects may not be garbage-collected. This is because the garbage collector of KCL does not take care of Lisp objects used in the C function. See the following C function which is assumed to return a two-element list consisting of its two arguments.

```
(defCfun ``object list2(x,y) object x,y;'' 0
%       object z;
        ('nil z)
        ((cons y z) z)
        ((cons x z) z)
 %      return(z);
 )
```

When invoked, `list2` first sets `nil` to the variable `z`, conses `y` to `z`, and then conses `x`. Each time `cons` is called, a new cons cell is allocated and the pointer to this cell is stored in `z`. However, there is no way to inform the garbage collector that the cells are referenced from the C variable `z`. Suppose that the cons cell allocated by the first `cons` is the last cons cell available at that time. Then, during execution of the second call to `cons`, the garbage collector begins to run and, unfortunately, the cons cell in `z` will be destroyed so that the cell can be recycled for further use.

To prevent a Lisp object from being unexpectedly garbage collected, the user must save the object in some place that is recognized by the garbage collector. The second parameter *n* to `defCfun` is used to reserve *n* such places for each call to the C function. In the body of the C function, these reserved places are referenced as `vs[0] , ... , vs[`*n-1*`]`. The function `list2` above, therefore, should be revised as follows.

```
(defCfun ``object list2(x,y) object x,y;'' 1
        ('nil ``vs[0]'')
        ((cons y ``vs[0]'') ``vs[0]'')
        ((cons x ``vs[0]'') ``vs[0]'')
%       Creturn(vs[0]);
)
```

Notice that `return` is replaced by `Creturn`. `Creturn` is similar to `return` except that `Creturn` releases the reserved places on return from the function. In the C code within a `defCfun` form, write "`Creturn(`*value*`);`" instead of "`return(`*value*`);`", and write "`Cexit;`" instead of "`return;`".

Again, a `defCfun` form has the above meaning only when it appears as a top-level form in a Lisp source file. Otherwise, the form expands to `nil`.

KCL converts a Lisp object into a C-language data by using the Common Lisp function `coerce`: For the C-type `int` (or `char`), the object is first coerced to a Lisp integer and the least significant 32-bit (or 8-bit) field is used as the C `int` (or `char`). For the C-type `float` (or `double`), the object is coerced to a short-float (or a long-float) and this value is used as the `C float` (or `double`). Conversion from a C data into a Lisp object is obvious: `C char, int, float,` and `double` become the equivalent Lisp character, fixnum, short-float, and long-float, respectively.
Here we list the complete syntax of `Clines, defentry,` and `defCfun` macro forms.

```
Clines-form:
        (Clines { string }* )

defentry-form:
        (defentry function-symbol
                ( { C-type }* )
                { C-function-name | ( { C-type | void } C-function-name ) } )

defCfun-form:
        (defCfun string non-negative-integer
                { string
                  | ( function-symbol { value }* )
                  | (( function-symbol  { value }* ) { place }* ) }  )

value:
place:
         { C-expr | ( C-type C-expr ) }

C-function-name:
C-expr:
         { string | symbol }

C-type:
         { object | int | char | float | double }
```

Chapter 11

The Editor

KCL/AOS is equipped with a screen editor FeCl2 (*F*ull-screen *E*ditor as a *C*ommon *L*isp *TOO*l). FeCl2 is an EMACS-like editor with facilities for Lisp coding. FeCl2 is invoked from KCL by the function `ed` and the result of editing can be passed to KCL directly. For the details of FeCl2 refer to *The FeCl2 Editor Reference Manual*.

`ed &optional` *filename* [*Function*]

`ed` invokes FeCl2 and sets the edit file of FeCl2 to *filename*. If the filetype of the file is not explicitly specified, then the *filename* is first merged into #`''.lsp''`.

The FeCl2 editor is not supported by other versions of KCL. The function `ed` of KCL/VAX, KCL/SUN, and KCL/UST calls the vi editor. If you hate vi, define your own `ed` function using the function `system` described in Chapter 8.

Appendix A. KCL Summary

The following table lists all symbols defined in KCL. Each line has the following form.

symbol [kind] remark

where *kind* is Function, Macro, Special (i.e., Special form name), Variable, Constant, Symbol, or Keyword. In the table, some symbols are labeled both as a macro and a special form name. This means that, although these symbols are defined to be a macro name in the *Common Lisp Reference Manual* , KCL treats them as if they were special forms (see Section 9.1). The pages in the remark refer to pages in this report.

`*`	[Function]
`*`	[Variable]
`**`	[Variable]
`***`	[Variable]
`+`	[Function]
`+`	[Variable]
`++`	[Variable]
`+++`	[Variable]
`-`	[Function]
`-`	[Variable]
`/`	[Function]
`/`	[Variable]
`//`	[Variable]
`///`	[Variable]
`/=`	[Function]
`1+`	[Function]
`1-`	[Function]
`<`	[Function]
`<=`	[Function]
`=`	[Function]
`>`	[Function]
`>=`	[Function]
`:abort`	[Keyword]
`abs`	[Function]
`acons`	[Function]
`acos`	[Function]
`acosh`	[Function]
`adjoin`	[Function]

adjust-array [Function]

:adjustable	[Keyword]
adjustable-array-p	[Function]
allocate	[Function] Added.
allocate-contiguous-pages	[Function] Added.
allocated-contiguous-pages	[Function] Added.
allocated-pages	[Function] Added.
allocated-relocatable-pages	[Function] Added.
allocate-relocatable-pages	[Function] Added.
alpha-char-p	[Function]
alphanumericp	[Function]
and	[Special, Macro]
append	[Function]
:append	[Keyword]
apply	[Function]
applyhook	[Function]
applyhook	[Variable]
apropos	[Function]
apropos-list	[Function]
aref	[Function]
:array	[Keyword]
array-dimension	[Function]
array-dimension-limit	[Constant]
array-dimensions	[Function]
array-element-type	[Function]
array-has-fill-pointer-p	[Function]
array-in-bounds-p	[Function]
array-rank	[Function]
array-rank-limit	[Constant]
array-row-major-index	[Function]
array-total-size	[Function]
array-total-size-limit	[Constant]
arrayp	[Function]
ash	[Function]
asin	[Function]
asinh	[Function]
assert	[Macro]
assoc	[Function]
assoc-if	[Function]
assoc-if-not	[Function]
atan	[Function]

atanh	[Function]
atom	[Function]
:b	[Keyword] Abbreviate Break Loop Command.
:backtrace	[Keyword] Break Loop Command.
:base	[Keyword]
bit	[Function]
bit-and	[Function]
bit-andc1	[Function]
bit-andc2	[Function]
bit-eqv	[Function]
bit-ior	[Function]
bit-nand	[Function]
bit-nor	[Function]
bit-not	[Function]
bit-orc1	[Function]
bit-orc2	[Function]
bit-vector-p	[Function]
bit-xor	[Function]
block	[Special]
:block	[Keyword] Added.
:blocks	[Keyword] Break Loop Command.
boole	[Function]
boole-1	[Constant]
boole-2	[Constant]
boole-and	[Constant]
boole-andc1	[Constant]
boole-andc2	[Constant]
boole-c1	[Constant]
boole-c2	[Constant]
boole-clr	[Constant]
boole-eqv	[Constant]
boole-ior	[Constant]
boole-nand	[Constant]
boole-nor	[Constant]
boole-orc1	[Constant]
boole-orc2	[Constant]
boole-set	[Constant]
boole-xor	[Constant]
both-case-p	[Function]
boundp	[Function]

break	[Function]
break-on-warnings	[Variable]
break-enable	[Variable] Added.
butlast	[Function]
by	[Function] Added.
bye	[Function] Added.
byte	[Function]
byte-position	[Function]
byte-size	[Function]
:c	[Keyword] Abbreviated Break Loop Command.
caaaar	[Function]
caaadr	[Function]
caaar	[Function]
caadar	[Function]
caaddr	[Function]
caadr	[Function]
caar	[Function]
cadaar	[Function]
cadadr	[Function]
cadar	[Function]
caddar	[Function]
cadddr	[Function]
caddr	[Function]
cadr	[Function]
call-arguments-limit	[Constant]
car	[Function]
:case	[Keyword]
case	[Special, Macro]
catch	[Special]
ccase	[Macro]
cdaaar	[Function]
cdaadr	[Function]
cdaar	[Function]
cdadar	[Function]
cdaddr	[Function]
cdadr	[Function]
cdar	[Function]
cddaar	[Function]
cddadr	[Function]
cddar	[Function]

cdddar	[Function]
cddddr	[Function]
cdddr	[Function]
cddr	[Function]
cdr	[Function]
ceiling	[Function]
cerror	[Function]
:c-file	[Keyword] Added.
char	[Function]
char-bit	[Function]
char-bits	[Function]
char-bits-limit	[Constant]
char-code	[Function]
char-code-limit	[Constant]
char-control-bit	[Constant]
char-downcase	[Function]
char-equal	[Function]
char-font	[Function]
char-font-limit	[Constant]
char-greaterp	[Function]
char-hyper-bit	[Constant]
char-int	[Function]
char-lessp	[Function]
char-meta-bit	[Constant]
char-name	[Function]
char-not-equal	[Function]
char-not-greaterp	[Function]
char-not-lessp	[Function]
char-super-bit	[Constant]
char-upcase	[Function]
char/=	[Function]
char<	[Function]
char<=	[Function]
char=	[Function]
char>	[Function]
char>=	[Function]
character	[Function]
characterp	[Function]
check-type	[Macro]
:circle	[Keyword]

cis	[Function]
clear-input	[Function] Different.
clear-output	[Function] Different.
clines	[Macro] Added.
close	[Function] Different.
clrhash	[Function]
code-char	[Function]
coerce	[Function]
commonp	[Function]
compilation-speed	[Symbol] Optimize Quality.
compile	[Function]
compile-file	[Function]
compiled-function-p	[Function]
compiler-let	[Special]
complex	[Function]
complexp	[Function]
:conc-name	[Keyword]
concatenate	[Function]
cond	[Special, Macro]
conjugate	[Function]
cons	[Function]
:console	[Keyword] Added.
consp	[Function]
constantp	[Function]
:constructor	[Keyword]
:copier	[Keyword]
copy-alist	[Function]
copy-list	[Function]
copy-readtable	[Function]
copy-seq	[Function]
copy-symbol	[Function]
copy-tree	[Function]
cos	[Function]
cosh	[Function]
count	[Function]
:count	[Keyword]
count-if	[Function]
count-if-not	[Function]
:create	[Keyword]
ctypecase	[Macro]

:current	[Keyword] Break Loop Command.
:data	[Keyword] Added.
:data-file	[Keyword] Added.
:debug	[Keyword] Added.
debug-io	[Variable]
decf	[Special, Macro]
declaration	[Symbol] Declaration Specifier.
declare	[Special]
decode-float	[Function]
decode-universal-time	[Function]
:default	[Keyword]
default-pathname-defaults	[Variable]
:defaults	[Keyword]
defautoload	[Macro] Added.
defcfun	[Macro] Added.
defconstant	[Macro]
defentry	[Macro] Added.
define-modify-macro	[Macro]
define-setf-method	[Macro]
defla	[Macro] Added.
defmacro	[Special, Macro]
defparameter	[Macro]
defsetf	[Macro]
defstruct	[Macro]
deftype	[Macro]
defun	[Special, Macro]
defvar	[Macro]
delete	[Function]
delete-duplicates	[Function]
delete-file	[Function]
delete-if	[Function]
delete-if-not	[Function]
denominator	[Function]
deposit-field	[Function]
describe	[Function]
:device	[Keyword]
digit-char	[Function]
digit-char-p	[Function]
:dir	[Keyword] Added.
:direction	[Keyword]

directory	[Function]
:directory	[Keyword]
directory-namestring	[Function]
disassemble	[Function] Different.
:displaced-index-offset	[Keyword]
:displaced-to	[Keyword]
do	[Special, Macro]
do*	[Special, Macro]
do-all-symbols	[Macro]
do-external-symbols	[Macro]
do-symbols	[Macro]
documentation	[Function]
dolist	[Special, Macro]
dotimes	[Special, Macro]
double-float-epsilon	[Constant]
double-float-negative-epsilon	[Constant]
dpb	[Function]
dribble	[Function]
ecase	[Macro]
ed	[Function] Different.
eighth	[Function]
:element-type	[Keyword]
elt	[Function]
encode-universal-time	[Function]
:end	[Keyword]
:end1	[Keyword]
:end2	[Keyword]
endp	[Function]
enough-namestring	[Function]
&environment	[Symbol] Defmacro-lambda Keyword.
eq	[Function]
eql	[Function]
equal	[Function]
equalp	[Function]
error	[Function]
:error	[Keyword]
error-output	[Variable]
:escape	[Keyword]
etypecase	[Macro]
eval	[Function]

evalhook	[Function]
evalhook	[Variable]
eval-when	[Macro]
eval-when-compile	[Variable] Added.
evenp	[Function]
every	[Function]
exp	[Function]
export	[Function]
expt	[Function]
:external	[Keyword]
:fasl-file	[Keyword] Added.
fboundp	[Function]
fceiling	[Function]
features	[Variable]
ffloor	[Function]
fifth	[Function]
file-author	[Function]
file-length	[Function]
file-namestring	[Function]
file-position	[Function]
file-write-date	[Function]
fill	[Function]
fill-pointer	[Function]
:fill-pointer	[Keyword]
find	[Function]
find-all-symbols	[Function]
find-if	[Function]
find-if-not	[Function]
find-package	[Function]
find-symbol	[Function]
finish-output	[Function]
first	[Function]
flet	[Special]
float	[Function]
float-digits	[Function]
float-precision	[Function]
float-radix	[Function]
float-sign	[Function]
floatp	[Function]
floor	[Function]

fmakunbound	[Function]
force-output	[Function]
format	[Function]
fourth	[Function]
fresh-line	[Function]
:from-end	[Keyword]
fround	[Function]
ftruncate	[Function]
ftype	[Symbol] Declaration Specifier.
funcall	[Function]
function	[Special] Also Declaration Specifier.
functionp	[Function]
:functions	[Keyword] Break Loop Command.
gbc	[Function] Added.
gcd	[Function]
gensym	[Function]
:gensym	[Keyword]
gentemp	[Function]
get	[Function]
get-decoded-time	[Function]
get-dispatch-macro-character	[Function]
get-internal-real-time	[Function]
get-internal-run-time	[Function]
get-macro-character	[Function]
get-output-stream-string	[Function]
get-properties	[Function]
get-setf-method	[Function]
get-setf-method-multiple-value	[Function]
get-universal-time	[Function]
getf	[Function]
gethash	[Function]
go	[Special]
graphic-char-p	[Function]
hash-table-count	[Function]
hash-table-p	[Function]
:h	[Keyword] Abbreviated Break Loop Command.
:help	[Keyword] Break Loop Command.
:h-file	[Keyword] Added.
:hide	[Keyword] Break Loop Command.
:hide-package	[Keyword] Break Loop Command.
:host	[Keyword]

host-namestring	[Function]
identity	[Function]
if	[Special]
:if-does-not-exist	[Keyword]
:if-exists	[Keyword]
ignore	[Symbol] Declaration Specifier.
ignore-maximum-pages	[Variable] Added.
imagpart	[Function]
import	[Function]
in-package	[Function]
incf	[Special, Macro]
:include	[Keyword]
:index	[Keyword]
:inherited	[Keyword]
:initial-contents	[Keyword]
:initial-element	[Keyword]
:initial-offset	[Keyword]
:initial-value	[Keyword]
inline	[Symbol] Declaration Specifier.
:input	[Keyword]
input-stream-p	[Function]
inspect	[Function]
int-char	[Function]
integer-decode-float	[Function]
integer-length	[Function]
integerp	[Function]
intern	[Function]
:intern	[Keyword]
internal-time-units-per-second	[Constant]
intersection	[Function]
:io	[Keyword]
:ioc	[Keyword] Added.
isqrt	[Function]
:junk-allowed	[Keyword]
:key	[Keyword]
keywordp	[Function]
:l	[Keyword] Abbreviated Break Loop Command.
labels	[Special]
lambda-list-keywords	[Constant]
lambda-parameters-limit	[Constant]

last	[Function]
last-termination-message	[Function] Added.
lcm	[Function]
ldb	[Function]
ldb-test	[Function]
ldiff	[Function]
least-negative-double-float	[Constant]
least-negative-long-float	[Constant]
least-negative-short-float	[Constant]
least-negative-single-float	[Constant]
least-positive-double-float	[Constant]
least-positive-long-float	[Constant]
least-positive-short-float	[Constant]
least-positive-single-float	[Constant]
length	[Function]
:length	[Keyword]
let	[Special]
let*	[Special]
:level	[Keyword]
lisp-implementation-type	[Function]
lisp-implementation-version	[Function]
list	[Function]
:list	[Keyword] Added.
list*	[Function]
list-all-packages	[Function]
list-length	[Function]
listen	[Function] Different.
listp	[Function]
load	[Function]
load-verbose	[Variable]
:local	[Keyword] Break Loop Command.
locally	[Special, Macro]
log	[Function]
logand	[Function]
logandc1	[Function]
logandc2	[Function]
logbitp	[Function]
logcount	[Function]
logeqv	[Function]
logior	[Function]

lognand	[Function]
lognor	[Function]
lognot	[Function]
logorc1	[Function]
logorc2	[Function]
logtest	[Function]
logxor	[Function]
long-float-epsilon	[Constant]
long-float-negative-epsilon	[Constant]
long-site-name	[Function]
loop	[Special, Macro]
lower-case-p	[Function]
machine-instance	[Function]
machine-type	[Function]
machine-version	[Function]
macro-function	[Function]
macroexpand	[Function]
macroexpand-1	[Function]
macroexpand-hook	[Variable]
macrolet	[Special]
make-array	[Function]
make-broadcast-stream	[Function]
make-char	[Function]
make-concatenated-stream	[Function]
make-dispatch-macro-character	[Function]
make-echo-stream	[Function]
make-hash-table	[Function]
make-list	[Function]
make-package	[Function]
make-pathname	[Function]
make-random-state	[Function]
make-sequence	[Function]
make-string	[Function]
make-string-input-stream	[Function]
make-string-output-stream	[Function]
make-symbol	[Function]
make-synonym-stream	[Function]
make-two-way-stream	[Function]
makunbound	[Function]
map	[Function]

mapc	[Function]
mapcan	[Function]
mapcar	[Function]
mapcon	[Function]
maphash	[Function]
mapl	[Function]
maplist	[Function]
mask-field	[Function]
max	[Function]
maximum-allocatable-pages	[Function] Added.
maximum-contiguous-pages	[Function] Added.
member	[Function]
member-if	[Function]
member-if-not	[Function]
merge	[Function]
merge-pathnames	[Function]
min	[Function]
minusp	[Function]
mismatch	[Function]
mod	[Function]
modules	[Variable]
most-negative-double-float	[Constant]
most-negative-fixnum	[Constant]
most-negative-long-float	[Constant]
most-negative-short-float	[Constant]
most-negative-single-float	[Constant]
most-positive-double-float	[Constant]
most-positive-fixnum	[Constant]
most-positive-long-float	[Constant]
most-positive-short-float	[Constant]
most-positive-single-float	[Constant]
multiple-value-bind	[Special, Macro]
multiple-value-call	[Special]
multiple-value-list	[Special, Macro]
multiple-value-prog1	[Special]
multiple-value-setq	[Special, Macro]
multiple-values-limit	[Constant]
:n	[Keyword] Abbreviated Break Loop Command.
:name	[Keyword]
name-char	[Function]

:named	[Keyword]
namestring	[Function]
nbutlast	[Function]
nconc	[Function]
:new-version	[Keyword]
:next	[Keyword] Break Loop Command.
nil	[Constant]
nintersection	[Function]
ninth	[Function]
not	[Function]
notany	[Function]
notevery	[Function]
notinline	[Symbol] Declaration Specifier.
nreconc	[Function]
nreverse	[Function]
nset-difference	[Function]
nset-exclusive-or	[Function]
nstring-capitalize	[Function]
nstring-downcase	[Function]
nstring-upcase	[Function]
nsublis	[Function]
nsubst	[Function]
nsubst-if	[Function]
nsubst-if-not	[Function]
nsubstitute	[Function]
nsubstitute-if	[Function]
nsubstitute-if-not	[Function]
nth	[Function]
nthcdr	[Function]
null	[Function]
numberp	[Function]
numerator	[Function]
nunion	[Function]
:ob-file	[Keyword] Added.
object	[Symbol] Declaration Specifier. Added.
oddp	[Function]
:o-file	[Keyword] Added.
open	[Function] Different.
optimize	[Symbol] Declaration Specifier.
or	[Special, Macro]

:output	[Keyword]
:output-file	[Keyword]
output-stream-p	[Function]
:overwrite	[Keyword]
package	[Variable]
package-name	[Function]
package-nicknames	[Function]
package-shadowing-symbols	[Function]
package-use-list	[Function]
package-used-by-list	[Function]
packagep	[Function]
pairlis	[Function]
:parent	[Keyword] Added.
parse-integer	[Function]
parse-namestring	[Function]
pathname	[Function]
pathname-device	[Function]
pathname-directory	[Function]
pathname-host	[Function]
pathname-name	[Function]
pathname-type	[Function]
pathname-version	[Function]
pathnamep	[Function]
peek-char	[Function]
phase	[Function]
pi	[Constant]
plusp	[Function]
pop	[Macro]
position	[Function]
position-if	[Function]
position-if-not	[Function]
pprint	[Function]
:p	[Keyword] Abbreviate Break Loop Command.
:predicate	[Keyword]
:preserve-whitespace	[Keyword]
:pretty	[Keyword]
:previous	[Keyword] Break Loop Command.
prin1	[Function]
prin1-to-string	[Function]

princ	[Function]
princ-to-string	[Function]
print	[Function]
:print	[Keyword]
print-array	[Variable]
print-base	[Variable]
print-case	[Variable]
print-circle	[Variable]
print-escape	[Variable]
:print-function	[Keyword]
print-gensym	[Variable]
print-length	[Variable]
print-level	[Variable]
print-pretty	[Variable]
print-radix	[Variable]
:probe	[Keyword]
probe-file	[Function]
process	[Function] Added.
proclaim	[Function]
proclamation	[Function] Added.
prog	[Special, Macro]
prog*	[Special, Macro]
prog1	[Special, Macro]
prog2	[Special, Macro]
progn	[Special]
progv	[Special]
provide	[Function]
psetf	[Macro]
psetq	[Special, Macro]
push	[Special, Macro]
pushnew	[Macro]
:q	[Keyword] Abbreviated Break Loop Command.
query-io	[Variable]
:quit	[Keyword] Break Loop Command.
quote	[Special]
:r	[Keyword] Abbreviated Break Loop Command.
:radix	[Keyword]
random	[Function]
random-state	[Variable]
random-state-p	[Function]
rassoc	[Function]

rassoc-if	[Function]
rassoc-if-not	[Function]
rational	[Function]
rationalize	[Function]
rationalp	[Function]
read	[Function]
read-base	[Variable]
read-byte	[Function] Different.
read-char	[Function]
read-char-no-hang	[Function] Different.
read-default-float-format	[Variable]
read-delimited-list	[Function]
read-from-string	[Function]
read-line	[Function]
:read-only	[Keyword]
read-preserving-whitespace	[Function]
read-suppress	[Variable]
readtable	[Variable]
readtablep	[Function]
realpart	[Function]
reduce	[Function]
:rehash-size	[Keyword]
:rehash-threshold	[Keyword]
rem	[Function]
remf	[Macro]
remhash	[Function]
remove	[Function]
remove-duplicates	[Function]
remove-if	[Function]
remove-if-not	[Function]
remprop	[Function]
:rename	[Keyword]
:rename-and-delete	[Keyword]
rename-file	[Function]
rename-package	[Function]
replace	[Function]
require	[Function]
:resume	[Keyword] Break Loop Command.
rest	[Function]
return	[Special, Macro]

return-from	[Special]
revappend	[Function]
reverse	[Function]
room	[Function]
:root	[Function] Added.
rotatef	[Macro]
round	[Function]
rplaca	[Function]
rplacd	[Function]
safety	[Symbol] Optimize Quality.
save	[Function] Added.
sbit	[Function]
scale-float	[Function]
schar	[Function]
search	[Function]
second	[Function]
set	[Function]
set-char-bit	[Function]
set-difference	[Function]
set-dispatch-macro-character	[Function]
set-exclusive-or	[Function]
set-macro-character	[Function]
set-syntax-from-char	[Function]
setf	[Special, Macro]
setq	[Special]
seventh	[Function]
shadow	[Function]
shadowing-import	[Function]
shiftf	[Macro]
short-float-epsilon	[Constant]
short-float-negative-epsilon	[Constant]
short-site-name	[Function]
signum	[Function]
simple-bit-vector-p	[Function]
simple-string-p	[Function]
simple-vector-p	[Function]
sin	[Function]
single-float-epsilon	[Constant]
single-float-negative-epsilon	[Constant]
sinh	[Function]

sixth	[Function]
:size	[Keyword]
sleep	[Function]
software-type	[Function]
software-version	[Function]
some	[Function]
sort	[Function]
space	[Symbol] Optimize Quality.
special	[Symbol] Declaration Specifier.
special-form-p	[Function]
speed	[Symbol] Optimize Quality.
sqrt	[Function]
stable-sort	[Function]
standard-char-p	[Function]
standard-input	[Variable]
standard-output	[Variable]
:start	[Keyword]
:start1	[Keyword]
:start2	[Keyword]
:static	[Keyword] Added.
step	[Macro]
:stream	[Keyword]
stream-element-type	[Function]
streamp	[Function]
string	[Function]
string-capitalize	[Function]
string-char-p	[Function]
string-downcase	[Function]
string-equal	[Function]
string-greaterp	[Function]
string-left-trim	[Function]
string-lessp	[Function]
string-not-equal	[Function]
string-not-greaterp	[Function]
string-not-lessp	[Function]
string-right-trim	[Function]
string-trim	[Function]
string-upcase	[Function]
string/=	[Function]
string<	[Function]

string<=	[Function]
string=	[Function]
string>	[Function]
string>=	[Function]
stringp	[Function]
sublis	[Function]
subseq	[Function]
subsetp	[Function]
subst	[Function]
subst-if	[Function]
subst-if-not	[Function]
substitute	[Function]
substitute-if	[Function]
substitute-if-not	[Function]
subtypep	[Function]
:supersede	[Keyword]
svref	[Function]
sxhash	[Function]
symbol-function	[Function]
symbol-name	[Function]
symbol-package	[Function]
symbol-plist	[Function]
symbol-value	[Function]
symbolp	[Function]
system	[Function] Added.
t	[Constant]
:tags	[Keyword] Break Loop Command.
tagbody	[Special]
tailp	[Function]
tan	[Function]
tanh	[Function]
tenth	[Function]
terminal-io	[Variable] Different.
termination-message	[Function] Added.
terpri	[Function]
:test	[Keyword]
:test-not	[Keyword]
the	[Special]
third	[Function]
throw	[Special]

time	[Macro]
trace	[Macro]
trace-output	[Variable]
tree-equal	[Function]
truename	[Function]
truncate	[Function]
type	[Symbol] Declaration Specifier.
:type	[Keyword]
type-of	[Function]
typecase	[Macro]
typep	[Function]
unexport	[Function]
unintern	[Function]
union	[Function]
:unhide	[Keyword] Break Loop Command.
:unhide-package	[Keyword] Break Loop Command.
unless	[Special, Macro]
unread-char	[Function]
untrace	[Macro]
unuse-package	[Function]
unwind-protect	[Special]
upper-case-p	[Function]
use-package	[Function]
user-homedir-pathname	[Function]
:username	[Keyword] Added.
:v	[Keyword] Abbreviated Break Loop Command.
values	[Function]
values-list	[Function]
:variables	[Keyword] Break Loop Command.
vector	[Function]
vector-pop	[Function]
vector-push	[Function]
vector-push-extend	[Function]
vectorp	[Function]
:verbose	[Keyword]
:version	[Keyword]
warn	[Function]
when	[Special, Macro]
&whole	[Symbol] Defmacro-lambda Keyword.
:wild	[Keyword] Added.
with-input-from-string	[Macro]

with-open-file	[Macro]	
with-open-stream	[Macro]	
with-output-to-string	[Macro]	
write	[Function]	
write-byte	[Function]	Different.
write-char	[Function]	
write-line	[Function]	
write-string	[Function]	
write-to-string	[Function]	
y-or-n-p	[Function]	
yes-or-no-p	[Function]	
zerop	[Function]	

Appendix B. An Overview of Kyoto Common Lisp

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 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. This document reports the current status of KCL, its implementation, and system performance. This document is a draft: the description is still incomplete and informal, and some technical terms are used without definition or explanation. A full paper on the KCL implementation is in preparation.

KCL is a full Common Lisp system.

KCL is a full implementation of the Common Lisp language described in the *Common Lisp Reference Manual*:

Common Lisp: The Language.
by Guy L. Steele et al.
Digital Press, 1984

KCL supports all Common Lisp functions, macros, and special forms defined in the *Common Lisp Reference Manual.* All Common Lisp variables and constants are defined in KCL exactly as described in the *Common Lisp Reference Manual.*

KCL is available on several machines already.

Currently, there are four major versions of KCL:

1. KCL/AOS Machine: Data General's Eclipse MV series super-minicomputers (MV10000, MV8000, MV6000, and MV4000) Operating System: Data General's original AOS/VS (Advanced Operating System / Virtual Storage)
2. KCL/VAX Machine: Digital Equipment Corporation's VAX 11 series machines (VAX 11/780 and VAX 11/750) Operating System: UNIX 4.2 bsd
3. KCL/SUN Machine: Sun Microsystems' Sun Workstation (MC68000 base) Operating System: UNIX 4.2 bsd

4. KCL/UST Machine: Sumitomo Electric Industries and Digital Computer Laboratory's personal workstation Ustation E15 (MC68000 base) Operating System: UNIX V (Uniplus' version)

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 are in preparation at other organizations. KCL is expected to become available on the following machines in the near future.

* IBM (and IBM-compatible) M series machines
* Apollo Domain
* Perkins Elmer
* VAX 11 (running VMS)
* VAX 11 (running Eunice, the Unix emulator)
* Eclipse MV (running DGUX, Data General's native Unix)

KCL is written in C and Lisp

The kernel of KCL is written in C, including:

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

The KCL compiler is entirely written in Common Lisp.

Each Common Lisp function or macro is written either in C or in Lisp.

```
in C:
        418 Common Lisp functions
        11 Common Lisp macros

in Lisp:
        133 Common Lisp functions
        59 Common Lisp macros
```

The size of the source code is:

```
        C code         705 Kbytes
Common Lisp functions and macros written 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.

KCL/AOS is built up by old-fashioned bootstrapping

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.

Objects are represented by cells.

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:

```
|----------|-----------|
|  `CONS'  |  mark-bit |
|----------|-----------|
|      car-pointer     |
|----------------------|
|      cdr-pointer     |
|----------------------|
```

and a fixnum cell consists of two words:

```
|----------|-----------|
| 'FIXNUM' |  mark-bit |
|----------|-----------|
|    fixnum-value      |
|----------------------|
```

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.

The heap is divided into pages.

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.

The actual configuration of the KCL heap is:

```
lower address                                              higher address
       | cell pages and binary pages | hole | relocatable pages |
```

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 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 like:

```
lower address                                              higher address
               | cell/binary pages and static pages |
```

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

KCL uses five stacks.

KCL uses the following stacks.

* Value Stack, for arguments/values passing lexical variables allocation 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 temporary values saving, in addition to the obvious use such as function invocation

Arguments/values are passed via the value stack.

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 like:

```
         |---------|
 vs_top ->|         |
```

```
          |---------|
          |    2    |
          |---------|
vs_base ->|    1    |
          |---------|
          |         |
          :         :
          |         |
   bottom |---------|
         value stack
```

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.

```
          |---------|
          |         |
          |---------|
 vs_top ->|         |
          |---------|
vs_base ->| (1 . 2) |
          |---------|
          |         |
          :         :
          |         |
   bottom |---------|
         value stack
```

Because the same stack pointers are used both for argument passing and for return value passing, the Common Lisp function values does almost nothing.

```
On the call of (values 1 2)    On return from (values 1 2)

          |---------|                    |---------|
 vs_top ->|         |           vs_top ->|         |
          |---------|                    |---------|
          |    2    |                    |    2    |
          |---------|                    |---------|
vs_base ->|    1    |          vs_base ->|    1    |
          |---------|                    |---------|
```

```
          |         |                |         |
          :         :                :         :
          |         |                |         |
   bottom |---------|         bottom |---------|
          value stack                value stack
```

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.

The interpreter uses A-lists.

The KCL interpreter uses three A-lists (Association lists) to represent lexical environments.

* One for variable bindings

* One for local function/macro definitions

* 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.

The invocation history stack is used for debugging.

The invocation history stack consists of two kinds of elements. Each element may be either a pair of a Lisp form and a pointer to lexical environment:

form	environment-pointer

or a pair of a function name and a pointer to the value stack:

function-name	value-stack-pointer

The former is pushed on the invocation history stack when an interpreted code is evaluated. The *form* is the interpreted code itself and the *environment-pointer* points to the three consecutive memory words each of which holds the A-list that represents the lexical environment. The latter is pushed when a compiled function is invoked. The *function-name* is the name of the called function and the *value-stack-pointer* points to the value stack location which is pointed to by vs_base when the function is called. For both kinds, the element on the invocation history stack is poped at the end of the evaluation.

Let us see how the invocation history stack is used for debugging.

```
>(defun fact (x)                ;;;  Wrong definition of the
   (if (= x 0)                  ;;;  factorial function.
       one                      ;;; one  should be  1 .
       (* x (fact (1- x)))))
fact

>(fact 3)                       ;;;  Tries  3!

Error: The variable ONE is unbound.
Error signalled by IF.

Broken at IF.
>>:b                            ;;;  Backtrace.
Backtrace: eval > fact > if > fact > if > fact > if > fact > IF
                                ;;;  Currently at the last  if.
>>:h                            ;;;  Help.

:c(urrent)              Show current function.
:p(revious)             Move to previous function.
:n(ext)                 Move to next function.
:b(acktrace)            Backtrace.
:h(elp)                 Help.
:q(uit)                 Return to top-level.
:r(esume)               Return to the caller
:l(ocal)                Show n-th local value.
:v(ariables)            Show local variables.
:functions              Show local functions.
:blocks                 Show block names.
:tags                   Show tags.
:hide                   Hide a function
:hide-package           Hide a package.
:unhide                 Unhide a function
:unhide-package         Unhide a package.
```

```
>>:p                        ;;;  Move to the last call of  fact.
Broken at FACT.

>>:b
Backtrace: eval > fact > if > fact > if > fact > if > FACT > if
                            ;;;  Now at the last  fact.
>>:v                        ;;;  The environment at the last call
Local variables: x.         ;;;  to  fact  is recovered.
                            ;;; x  is the only bound variable.
>>x
0                           ;;;  The value of  x  is  0.

>>:blocks
Block names: fact.          ;;;  The block  fact  is established.

>>(return-from fact 1)      ;;;  Return from the last call of
6                           ;;; fact  with the value of  0.
                            ;;;  The execution is resumed and
>                           ;;;  the value  6  is returned.
                            ;;;  Again at the top-level loop.
```

The KCL compiler generates intermediate code in C.

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 is 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 compiler mimics human C programmer.

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 following macro definition is found in the H-file.

```
#define VM3 2\\
```

When the compiled add1 is called, the value stack looks like:

```
On the call of (add1 1)    On return from (add1 1)

          |---------|                |---------|
          |         |      vs_top -> |         |
          |---------|                |---------|
 vs_top ->|         |      vs_base ->|    2    |
          |---------|                |---------|
vs_base ->|    1    |                |    1    |
          |---------|                |---------|
          |         |                |         |
          :         :                :         :
          |         |                |         |
   bottom |---------|         bottom |---------|
           value stack                value stack
```

Note that the two value stack pointers need not be moved in this example. This shows that the KCL compiler still has room for improvement.

Lexical environment of compiled closures is represented by a list.

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 the "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 popped 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

        >
```

After this, the two compiled closures look like:

```
second closure       y:                     x:
|-------|------|      |-------|------|       |------|------|
|  **   |    --|----->|  10   |    --|------>|  21  | nil  |
```

```
|-------|------|      |-------|------|       |------|------|
                                                   ^
                     first closure                 |
                     |-------|------|              |
                     |   *   |    --|----------|
                     |-------|------|
```

```
 * : address of the compiled code for #'(lambda () (incf x))
** : address of the compiled code for #'(lambda () (incf x y))
```

Declarations increase code efficiency.

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.

KCL does not have "disassembler".

Common Lisp defines a function disassemble , which is supposed to disassemble a compiled function and to display the assembler code. According to the *Common Lisp Reference Manual*,

> *This is primary useful for debugging the compiler, ..*

This is, however, *useless* in our case, because we are not concerned with assembly language. Rather, we are interested in the C code generated by the KCL compiler. Thus the disassemble function in KCL accepts not-yet-compiled functions only and displays the translated C code.

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

>(disassemble 'add1)
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_check;
        vs_top=sup;
        base[1]=one_plus(base[0]);
        vs_top=(vs_base=base+1)+1;
        return;
}

>
```

KCL/AOS has its own screen editor.

KCL/AOS has an embedded full-screen editor which resembles EMACS. The editor is called FeCl2 as the acronym of Full-screen Editor as a Common Lisp TOOl. It is invoked by the Common Lisp function `ed`. Unfortunately, FeCl2 is not supported by other versions of KCL, simply because we are too lazy to port it (it is written in C). For these versions, `ed` invokes the `vi` editor of Unix.

KCL has C language interface.

The user can embed his or her own C code into Lisp source code. The idea is very simple: The specified C code is inserted in the intermediate C code that is generated by the KCL compiler. In the following example, `Clines` and `defentry` are top-level macros specific to KCL. The `Clines` macro form specifies the C code to be embedded, in terms of strings, and the `defentry` form defines an entry of the specified C function from KCL.

```
(Clines
"   int tak(x, y, z)                        "
"   int x, y, z;                            "
"   {   if (y >= x) return(z);              "
"       else return(tak(tak(x-1, y, z),     "
"                       tak(y-1, z, x),     "
"                       tak(z-1, x, y)));   "
"   }                                       "
)

(defentry tak (int int int) (int "tak"))
```

Port to VAX took three days.

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. The following system parameters were redefined because Vax's representation of floating point numbers differs from that of Eclipse.

   ```
   most-positive-short-float
   most-negative-short-float
   least-positive-short-float
   least-negative-short-float
   most-positive-long-float
   most-positive-double-float
   most-positive-single-float
   most-negative-long-float
   most-negative-double-float
   most-negative-single-float
   least-positive-long-float
   least-positive-double-float
   least-positive-single-float
   least-negative-long-float
   least-negative-double-float
   ```

```
least-negative-single-float
short-float-epsilon
short-float-negative-epsilon
long-float-epsilon
double-float-epsilon
single-float-epsilon
long-float-negative-epsilon
double-float-negative-epsilon
single-float-negative-epsilon
```

4. For the same reason as above, the following machine-dependent Common Lisp functions were rewritten.

```
decode-float
scale-float
float-radix
float-digits
float-precision
integer-decode-float
```

5. The three assembler routines were rewritten.

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

7. The memory dump routine was rewritten.

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 took three evenings.

The port to the SUN Workstation was much easier than the port to the VAX, mainly because the operating system is the same for both VAX and SUN.

1. The compiler top-level of KCL/VAX was used without changes.

2. The file system of KCL/VAX was used without changes.

3. The system parameters that depend on the representation of floating point numbers were rewritten.

4. The Common Lisp functions that depend on the representation of floating point numbers were redefined.

5. The three assembler routines were rewritten.

6. The in-core loader of KCL/VAX was used without changes.
7. The memory dump routines of KCL/VAX was used without changes.

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.

Port to Ustation took one week.

The port to the Ustation was relatively a hard job. It took almost a week. The major difficulity was that the C compiler of Unix V on the Ustation recognized identifiers only by the first seven characters. As already mentioned, we used the convention that the C function that implements the Common Lisp function *function-name* is given the name `L` *function-name* . Thus, for example, the Common Lisp functions on packages such as

```
package-name
package-nickname
package-shadowing-symbols
package-use-list
package-used-by-list
packagep
```

are implemented by the C functions whose names all begin with Lpackage . These C functions are regarded as having the same name by the C compiler. This problem was solved by preparing a preprocessor program which maps long identifiers into smaller ones. This program is now used by the KCL compiler on Ustation before the C compiler is called.

KCL is relatively compact.

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).

Gabriel's benchmark.

The following table shows the results of Richard Gabriel's Lisp benchmark tests with the four versions of KCL. The results with five other Common Lisp systems are also listed for comparison. Each number represents the CPU time (in seconds) for the compiled program. ' * ' indicates that the time includes garbage-collection time. The data of S-1 Lisp and Spice Lisp are found in:

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

We received the data of Symbolics, DEC Common Lisp, and DG Common Lisp directly from Dr. Richard Gabriel in April 1985. We measured the data for KCL in July 1985. For the details of the benchmark tests, refer to the book above.

Benchmark Test	Boyer	Browse	Destruct	Traverse Initialize	Traverse Run
Kyoto CL MV10000	11.02	18.09	3.15	5.75	44.16
Kyoto CL VAX 780	43.73	68.15*	8.00	14.42	151.20
Kyoto CL E15	43.60*	73.08*	10.23	20.38	118.60
Kyoto CL SUN	47.00	81.37	14.10	26.30	160.58
Symbolics 3600	11.99	30.8	3.03	8.62	49.95
S-1 CL Mark IIA	10.03	10.2	0.91	1.93	30.1
Spice Perq	134.79	359.63	17.78	41.75	490.6
DEC CL VAX 780	46.79	118.51	6.38	20.76	161.68
DG CL MV10000	29.3	59.91	6.95	27.77	45.86

Benchmark Test	Tak	Stak	Ctak	Takl	Takr
Kyoto CL MV10000	0.42	1.90	4.35	5.13	0.54
Kyoto CL VAX 780	1.45	6.03	14.02	19.97	1.75
Kyoto CL E15	1.17	7.63	9.58	16.10	1.38
Kyoto CL SUN	1.48	11.02	16.88	23.35	1.67
Symbolics 3600	0.6	2.58	7.65	6.44	0.6
S-1 CL Mark IIA	0.29	4.31	0.82	2.92	0.58
Spice Perq	4.7	13.5	8.4	24.0	7.7
DEC CL VAX 780	1.83	4.11	8.09	7.34	3.42
DG CL MV10000	0.89	3.09	1.79	5.52	1.21

Benchmark Test	Deriv	DDeriv	Div2 Iterative	Div2 Recursive	FFT
Kyoto CL MV10000	4.85	5.93	1.99	2.73	1.55
Kyoto CL VAX 780	18.98	23.07	7.48	11.25	9.07
Kyoto CL E15	22.07*	25.38*	9.95*	12.60*	74.62
Kyoto CL SUN	20.72	24.77	9.32	12.30	94.07
Symbolics 3600	5.12	5.24	1.85	2.89	4.75
S-1 CL Mark IIA	4.99	3.27	0.82	1.49	1.44
Spice Perq	71.8	77.7	28.15	40.69	59.0
DEC CL VAX 780	13.76	—	5.0	9.84	32.69
DG CL MV10000	5.6	8.11	2.8	4.48	62.78

Benchmark Test	Puzzle	Triang	Fprint	Fread	Tprint
Kyoto CL MV10000	6.76	104.85	1.98	2.45	1.74
Kyoto CL VAX 780	20.57	366.12	5.73	5.77	6.55
Kyoto CL E15	41.30	341.23*	9.83	6.07	11.58
Kyoto CL SUN	49.13	499.68	10.05	7.62	8.50
Symbolics 3600	13.89	151.7	2.6	4.6	4.9
S-1 CL Mark IIA	1.82	62.06	—	—	—
Spice Perq	75.14	1488.85	20.0	26.0	22.6
DEC CL VAX 780	47.48	360.85	3.94	7.24	2.85
DG CL MV10000	138.2	151.2	2.35	4.65	2.83

Appendix C. Kyoto Common Lisp Installation Guide

This appendix explains how to install the KCL system, separately for each version of KCL.

C.1. Installation of KCL/AOS

1. Prepare a directory (hereafter called *KCL directory*) for Kyoto Common Lisp. In the following examples, we suppose that the KCL directory is `:UDD:KCL`.

2. Load the distribution tape to the KCL directory.

```
) DIR
:UDD:KCL
) LOAD
```

You will find that the subdirectory PORT has been created. The PORT subdirectory contains everything that is needed to run KCL. Files in this directory are:

```
Documents:
    README                  this file

Executable files:
    KCL.PR                  the KCL interpreter and compiler
    KCL.ST
    BUILD_FASL.PR           the fasl-file builder
    FECL2.PR                the FeCl2 editor

Command files:
    KCL.CLI                 to invoke KCL
    LC.CLI                  to invoke the KCL compiler
    LC1.CLI                 to invoke the KCL compiler
    FECL2.CL1               to invoke the FeCl2 editor

Header file:
    CMPINCLUDE.H            the header file for the KCL compiler

Miscellaneous:
    FECL2.CMD               FeCl2 editor command table
```

3. Customize the command files.

The command file KCL.CLI consists of the following command lines to invoke the KCL interpreter.

```
push;prom pop
sea :usr:dgc [!sea]
x :udd:[!user]:port:%0-% :udd:[!user]:port:
pop
```

Replace the two occurrences of " [!user] " with the name of the KCL directory (i.e., KCL in our example).

```
push;prom pop
sea :usr:dgc [!sea]
x :udd:KCL:port:%0-% :udd:KCL:port:
pop
```

And move this file to an appropriate command directory, say :UTIL, so that all KCL users can access it.

If the name of the directory that includes DG's C compiler is not :USR:DGC, you should replace it with the appropriate directory name, so that the KCL compiler can access the C compiler by CC.PR.

You can use the commands `LC.CLI` and `LC1.CLI` to invoke the compiler directly from the CLI. The content of `LC.CLI` is:

```
push
 prompt pop level
 sea :usr:dgc [!sea]
 WRITE Compiling %1%.LSP.
  proc/def/ioc/block/pri=3 \&
  :udd:[!user]:port:kcl :udd:[!user]:port: \%1\% \%1\% U10000
pop
```

Customize these command files in the same way as for the KCL.CLI command file above.

4. Install the header file

Copy the header file `CMPINCLUDE.H` to the standard directory for C include files, say `:USR:DGC`.

C.2. Installation of KCL/VAX

1. Prepare a directory (hereafter called *KCL directory*) for Kyoto Common Lisp. In the following examples, we suppose that the KCL directory is `/usr/kcl`.

2. Load the distribution tape to the KCL directory.

```
% pwd
usr/kcl
% tar x
```

You will find that the subdirectory `unixport` has been created. The `unixport` subdirectory contains everything that is needed to run KCL and the KCL compiler. Files in this directory are:

```
Documents:
   readme          this file

Executable files:
   saved_kcl       the KCL interpreter and compiler

Command files:
   kcl             to invoke KCL
   lc              to invoke the KCL compiler
   lc1             to invoke the KCL compiler

Header file:
   cmpinclude.h    the header file for the KCL compiler
```

3. Customize the command files.

The command file kcl consists of a single line command to invoke the KCL interpreter.

```
#
~/unixport/saved_kcl ~/unixport/
```

Replace two '   's with the pathname of the KCL directory (i.e., /usr/kcl in our example).

```
#
/usr/kcl/unixport/saved_kcl /usr/kcl/unixport/
```

And move this file to an appropriate command directory, say `/usr/bin`, so that all KCL users can access it.

You can use the commands lc and lc1 to invoke the compiler directly from the shell. The content of lc is:

```
#
echo Compiling $1.lsp
~/unixport/saved_kcl ~/unixport/ $1 $1 U1000
```

Customize these command files in the same way as for the kcl command file above.

4. Install the header file

Copy the header file `cmpinclude.h` to the standard directory for C include files, say `/usr/include`.

C.3. Installation of KCL/SUN

1. Prepare a directory (hereafter called *KCL directory*) for Kyoto Common Lisp. In the following examples, we suppose that the *KCL directory* is `/usr/kcl`.

2. Load the distribution tape to the KCL directory.

```
% pwd
/usr/kcl
% tar x
```

You will find that the subdirectory `unixport` has been created. The `unixport` subdirectory contains everything that is needed to run KCL and the KCL compiler. Files in this directory are:

```
Documents:
    readme            this file

Executable files:
    saved_kcl         the KCL interpreter and compiler

Command files:
    kcl               to invoke KCL
    lc                to invoke the KCL compiler
    lc1               to invoke the KCL compiler

Header file:
    cmpinclude.h      the header file for the KCL compiler
```

3. Customize the command files.

The command file `kcl` consists of a single line command to invoke the KCL interpreter.

```
#
~/unixport/saved_kcl ~/unixport/
```

Replace two ‘ ’s with the pathname of the KCL directory (i.e., /usr/kcl in our example).

```
#
/usr/kcl/unixport/saved_kcl /usr/kcl/unixport/
```

And move this file to an appropriate command directory, say `/usr/bin`, so that all KCL users can access it.

You can use the commands lc and lc1 to invoke the compiler directly from the shell. The content of lc is:

```
#
echo Compiling $1.lsp
~/unixport/saved_kcl ~/unixport/ $1 $1 U1000
```

Customize these command files in the same way as for the kcl command file above.

4. Install the header file

Copy the header file `cmpinclude.h` to the standard directory for C include files, say `/usr/include`.

C.4. Installation of KCL/UST

1. Prepare a directory (hereafter called *KCL directory*) for Kyoto Common Lisp. In the following examples, we suppose that the KCL directory is `/usr/kcl`.

2. Three floppy disks are used for the distribution. Load the floppy disks to the KCL directory by the multi-volume option of the tar command. (Insert the first disk to the drive and issue the tar command. Then, you will be prompted to change the disks.)

```
% pwd
/usr/kcl
% tar xvfBM /dev/rdy0g 1976
```

You will find that the subdirectory `unixport` has been created. The `unixport` subdirectory contains everything that is needed to run KCL and the KCL compiler. Files in this directory are:

```
Documents:
    readme              this file

Executable files:
    saved_kcl           the KCL interpreter and compiler
    ild                 incremental loader
    trans               translator from Unix 4.2 to Unix V

Command Source files:
    kcl.c               to invoke KCL
    lc.c                to invoke the KCL compiler
```

```
    lc1.c             to invoke the KCL compiler

Header file:
    cmpinclude.h      the header file for the KCL compiler
```

3. Customize the command files.

The source file `kcl.c` consists of a C-language program to invoke the KCL interpreter. Replace all `/usr/ukcl` in this file with the pathname of the KCL directory (i.e., `/usr/kcl` in our example), and compile this file.

```
% cc -o kcl kcl.c
```

And move the object file to an appropriate command directory, say `/usr/bin` , so that all KCL users can access it.

You can use the source files `lc.c` and `lc1.c` to invoke the compiler directly from the shell. Customize these source files in the same way as for the kcl command file above.

4. Install the header file

Copy the header file `cmpinclude.h` to the standard directory for C include files, say `/usr/include`.