Kyoto Common Lisp Report

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

August 1985
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 difference 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. Toshiyuki 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. Sief 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 Doshiita 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

1 How to Start and End a KCL Session ........................................... 2

2 Data Types .............................................................................. 5
  2.1 Numbers ............................................................................. 5
    2.1.1 Integers ........................................................................... 5
    2.1.2 Ration ............................................................................. 6
    2.1.3 Floating-Point Numbers ................................................. 6
    2.1.4 Complex Numbers ....................................................... 8
  2.2 Characters ............................................................................ 8
    2.2.1 Standard Characters .................................................... 8
    2.2.2 Line Divisions ............................................................. 8
    2.2.3 Non-standard Characters .............................................. 9
    2.2.4 Character Attributes ................................................. 9
    2.2.5 String Characters .................................................... 9
  2.3 Symbols ................................................................................. 9
  2.4 Lists and Conses ................................................................. 9
  2.5 Arrays ................................................................................ 9
    2.5.1 Vectors ........................................................................... 10
    2.5.2 Strings .......................................................................... 10
    2.5.3 Bit-Vectors ................................................................. 10
  2.6 Hash Tables ......................................................................... 10
  2.7 Readables ............................................................................. 10
  2.8 Packages ............................................................................. 10
  2.9 Pathnames ........................................................................... 11
  2.10 Streams ............................................................................. 13
  2.11 Random-States ................................................................. 14
  2.12 Structures ........................................................................... 14
  2.13 Functions ........................................................................... 14
  2.14 Unreadable Data Objects ................................................ 15
  2.15 Overlap, Inclusion, and Disjointness of Types ..................... 15
3 Input and Output 16
   3.1 Read Macros ........................................ 16
   3.2 Input and Output Functions ....................... 16

4 Memory Management 19
   4.1 Implementation Types .......................... 19
   4.2 Heap and Relocatable Areas .................. 21
   4.3 The Garbage Collector .......................... 22
   4.4 Allocation Functions ........................... 23
   4.5 Storage Information ............................. 24

5 Debugging Facilities 27
   5.1 The Tracer ........................................ 27
   5.2 The Stepper ....................................... 28
   5.3 Errors ............................................. 28
   5.4 The Break Loop ................................. 29
   5.5 Describe and Inspect ............................ 34

6 The Compiler 36

7 Declarations 40
   7.1 Declaration Specifiers .......................... 41
   7.2 Significant Type Specifiers .................... 46
   7.3 Treatment of Type Declarations ............... 47
      7.3.1 Variable Allocations ....................... 48
      7.3.2 Built-in Functions that Operate on Raw Data Directly 49
      7.3.3 Arguments/Values Passing .................. 51

8 Operating System Interface 53

9 Macros 55
   9.1 System Macros ................................... 55
   9.2 Defmacro Lambda-Lists ......................... 55

10 The C Language Interface 58

11 The Editor 65

A KCL Summary 66
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

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.
%

%
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 `Ctrl-C` or by typing DELETE) is a kind of error. It breaks the running program and calls the break level loop.

Example:

```lisp
(defun foo () (do () (nil)))
foo

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

Broken at F00.
>>
```
Chapter 2

Data Types


2.1 Numbers

2.1.1 Integers

Fixnums in KCL are those integers in the range $-2^{31}$ to $2^{31} - 1$, inclusive. Other integers are bignums. Thus 25 factorial (25!)

$$1551121004330985984000000$$

is certainly a bignum in KCL.

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

- `most-positive-fixnum` = 2147483647 = $2^{31} - 1$
- `most-negative-fixnum` = -2147483648 = $-2^{31}$

- `boole-1` = 3
- `boole-2` = 5
- `boole-and` = 1
- `boole-andc1` = 4
- `boole-andc2` = 2
- `boole-c1` = 12
- `boole-c2` = 10
- `boole-clr` = 0
bole-seqv = 9
bole-lor = 7
bole-nand = 14
bole-nor = 8
bole-orc1 = 13
bole-orc2 = 11
bole-set = 15
bole-xor = 6


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 1.0f0 1.0d0) is false, but (eql 1.0f0 1.0d0) is true. Similarly, (typep 1.0f0 'short-float) is false, but (typep 1.0f0 '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:

<table>
<thead>
<tr>
<th>Format</th>
<th>KCL/AOS</th>
<th>KCL/VAX</th>
<th>KCL/SUN</th>
<th>KCL/UST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short</td>
<td>24 bits</td>
<td>23 bits</td>
<td>24 bits</td>
<td>24 bits</td>
</tr>
<tr>
<td>Single</td>
<td>56 bits</td>
<td>55 bits</td>
<td>53 bits</td>
<td>53 bits</td>
</tr>
<tr>
<td>Double</td>
<td>56 bits</td>
<td>55 bits</td>
<td>53 bits</td>
<td>53 bits</td>
</tr>
<tr>
<td>Long</td>
<td>56 bits</td>
<td>55 bits</td>
<td>53 bits</td>
<td>53 bits</td>
</tr>
</tbody>
</table>

The floating-point exponent sizes are:

<table>
<thead>
<tr>
<th>Format</th>
<th>KCL/AOS</th>
<th>KCL/VAX</th>
<th>KCL/SUN</th>
<th>KCL/UST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short</td>
<td>7 bits</td>
<td>8 bits</td>
<td>8 bits</td>
<td>8 bits</td>
</tr>
<tr>
<td>Single</td>
<td>7 bits</td>
<td>8 bits</td>
<td>11 bits</td>
<td>11 bits</td>
</tr>
<tr>
<td>Double</td>
<td>7 bits</td>
<td>8 bits</td>
<td>11 bits</td>
<td>11 bits</td>
</tr>
<tr>
<td>Long</td>
<td>7 bits</td>
<td>8 bits</td>
<td>11 bits</td>
<td>11 bits</td>
</tr>
</tbody>
</table>

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
= 7.2370057732264f75 (KCL/AOS)
= 1.701411334604692f38 (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)
= 2.938735873055719f-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.

<table>
<thead>
<tr>
<th>Character</th>
<th>Code (in octal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\Space</td>
<td>040</td>
</tr>
<tr>
<td>\Newline</td>
<td>012</td>
</tr>
<tr>
<td>\Backspace</td>
<td>010</td>
</tr>
<tr>
<td>\Tab</td>
<td>011</td>
</tr>
<tr>
<td>\Linefeed</td>
<td>012</td>
</tr>
<tr>
<td>\Page</td>
<td>014</td>
</tr>
<tr>
<td>\Return</td>
<td>015</td>
</tr>
<tr>
<td>\Rubout</td>
<td>177</td>
</tr>
</tbody>
</table>

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.

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:

\[
\begin{align*}
\text{char-bits-limit} &= 1 \\
\text{char-code-limit} &= 256 \\
\text{char-control-bit} &= 0 \\
\text{char-font-limit} &= 1 \\
\text{char-hyper-bit} &= 0 \\
\text{char-meta-bit} &= 0 \\
\text{char-super-bit} &= 0
\end{align*}
\]


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 22.16 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.

<table>
<thead>
<tr>
<th>Array Type</th>
<th>Element Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(array t) and (vector t)</td>
<td>a cell pointer</td>
</tr>
<tr>
<td>(array fixnum) and (vector fixnum)</td>
<td>32 bit signed integer</td>
</tr>
<tr>
<td>(array string-char) and string</td>
<td>8 bit code</td>
</tr>
<tr>
<td>(array short-float) and (vector short-float)</td>
<td>32 bit floating point</td>
</tr>
<tr>
<td>(array long-float) and (vector long-float)</td>
<td>64 bit floating point</td>
</tr>
<tr>
<td>(array bit) and bit-vector</td>
<td>1 bit</td>
</tr>
</tbody>
</table>

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 `#\langle package-name package\rangle`.

### 2.9 Pathnames

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

```lisp
#"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 "demo")
  name: "queen"
  type: "lsp"

"../../demo/queen.lsp"
  directory: (: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 rep-
resenting 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

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 \texttt{(make-synonym-stream symbol )}.

\texttt{<a concatenated stream>}

An input stream generated by the function \texttt{make-concatenated-stream}.

\texttt{<a broadcast stream>}

An output stream generated by the function \texttt{make-broadcast-stream}.

\section*{2.11 Random-States}

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

\section*{2.12 Structures}

There are no implementation-dependent features for structures.

\section*{2.13 Functions}

An interpreted function (including macro expansion functions) is represented in one of the following formats.

\texttt{(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 \texttt{'(lambda \lambda-list . body)} is evaluated.

\texttt{(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 \texttt{(defun function-name \lambda-list . body)} is evaluated. In this case, \texttt{block-name} is identical to \texttt{function-name}.

\texttt{(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 \texttt{'(lambda \lambda-list . body)} (or, equivalently, \texttt{(function (lambda \lambda-list . body))}) is evaluated. \texttt{env1}, \texttt{env2}, and \texttt{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 fle, 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.

```lisp
#" 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.

```lisp
'#,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 fast file (see Chapter 6). If it fails, then load tries to load a file with the filetype .lisp. KCL assumes that .lisp 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 fast file if and only if the filetype of the file is .o. Other files are assumed to be source files.

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.

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.

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.

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.
<table>
<thead>
<tr>
<th>Implementation Type</th>
<th>Common Lisp Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>cons</td>
<td>cons</td>
</tr>
<tr>
<td>fixnum</td>
<td>fixnum</td>
</tr>
<tr>
<td>bignum</td>
<td>bignum</td>
</tr>
<tr>
<td>ratio</td>
<td>ratio</td>
</tr>
<tr>
<td>short-float</td>
<td>short-float</td>
</tr>
<tr>
<td>long-float</td>
<td>long-float (= double-float = single-float)</td>
</tr>
<tr>
<td>complex</td>
<td>complex</td>
</tr>
<tr>
<td>character</td>
<td>character</td>
</tr>
<tr>
<td>symbol</td>
<td>symbol</td>
</tr>
<tr>
<td>package</td>
<td>package</td>
</tr>
<tr>
<td>hash-table</td>
<td>hash-table</td>
</tr>
<tr>
<td>array</td>
<td>(and array (not vector))</td>
</tr>
<tr>
<td>vector</td>
<td>(and vector (not string) (not bit-vector))</td>
</tr>
<tr>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>bit-vector</td>
<td>bit-vector</td>
</tr>
<tr>
<td>structure</td>
<td>structure</td>
</tr>
<tr>
<td>stream</td>
<td>stream</td>
</tr>
<tr>
<td>random-state</td>
<td>random-state</td>
</tr>
<tr>
<td>readable</td>
<td>readable</td>
</tr>
<tr>
<td>cfun</td>
<td>compiled-function without environment</td>
</tr>
<tr>
<td>cclosure</td>
<td>compiled-function with environment</td>
</tr>
<tr>
<td>spice</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 4.1: Implementation Types

<table>
<thead>
<tr>
<th>Class</th>
<th>Implementation Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cons bignum ratio long-float complex</td>
</tr>
<tr>
<td>2</td>
<td>fixnum short-float character random-state readable spice</td>
</tr>
<tr>
<td>3</td>
<td>symbol package</td>
</tr>
<tr>
<td>4</td>
<td>array hash-table vector bit-vector stream pathname cclosure</td>
</tr>
<tr>
<td>5</td>
<td>string cfun</td>
</tr>
<tr>
<td>6</td>
<td>structure</td>
</tr>
</tbody>
</table>

Table 4.2: Classification of Implementation Types
<table>
<thead>
<tr>
<th>Type</th>
<th>Body</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>readtable</td>
<td>read</td>
<td>table</td>
</tr>
<tr>
<td>symbol</td>
<td>symbol</td>
<td>name</td>
</tr>
<tr>
<td>package</td>
<td>hash</td>
<td>table</td>
</tr>
<tr>
<td>array</td>
<td>array</td>
<td>body</td>
</tr>
<tr>
<td>hash-table</td>
<td>hash</td>
<td>table</td>
</tr>
<tr>
<td>vector</td>
<td>vector</td>
<td>body</td>
</tr>
<tr>
<td>bit-vector</td>
<td>bit-vector</td>
<td>body</td>
</tr>
<tr>
<td>stream</td>
<td>I/O buffer</td>
<td></td>
</tr>
<tr>
<td>closure</td>
<td>code</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>string</td>
<td>body</td>
</tr>
<tr>
<td>cfun</td>
<td>code</td>
<td></td>
</tr>
<tr>
<td>structure</td>
<td>structure</td>
<td>body</td>
</tr>
</tbody>
</table>

Table 4.3: Types with Bodies

### 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 \texttt{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 \texttt{allocate-contiguous-pages}. The size of the relocatable area is specified by the KCL specific function \texttt{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.

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** $x$  

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**  

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**  

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

**maximum-allocatable-pages type**  

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

**allocate-contiguous-pages number**  

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

**allocated-contiguous-pages**  

Returns the number of pages allocated for contiguous blocks.

**maximum-contiguous-pages**  

Returns the current maximum number of pages for contiguous blocks.

**allocate-relocatable-pages number**  

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**
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:

```lisp
>(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 FUCALL.
>>>(allocate 'cons 1000)
t
```

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
                  readable      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

\[ \text{level} > (\text{name} \ \text{arg}_1 \ldots \ \text{arg}_n) \]

On return from a traced function, it prints

\[ < \text{level} (\text{name} \ \text{value}_1 \ldots \ \text{value}_n) \]

\text{name} is the name of the traced function, \text{args} are the arguments, and \text{values} are the return values. \text{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 \text{level}.

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

\texttt{trace \{function-name\}*} \hspace{1cm} [Macro]

Causes one or more functions to be traced. \texttt{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 \texttt{notinline} declaration for the called function (see Chapter 7). \texttt{trace} returns a name list of those functions that were traced by the call to trace. If no \texttt{function-name} is given, \texttt{trace} simply returns a name list of all the currently traced functions.

\texttt{untrace \{function-name\}*} \hspace{1cm} [Macro]

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

\section*{5.2 The Stepper}

\begin{flushright}
\texttt{step form}
\end{flushright}

\texttt{[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 \texttt{*print-level*} and \texttt{*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 \texttt{n} was abbreviated.

The stepper commands are:

\begin{itemize}
  \item \texttt{n} \quad Next. Evaluates the current form in the single-step mode.
  \item \texttt{s} \quad Skip. Evaluates the current form in the ordinary mode. The single-step mode will be resumed at completion of the evaluation.
  \item \texttt{p} \quad Print. Pretty-prints the current form and then prompts again.
  \item \texttt{f fn} \quad Function. Evaluates the current form in the ordinary mode until the specified function \texttt{fn} is invoked. If the specified function is not invoked at all, then this command has the same effects as the \texttt{q} command below.
  \item \texttt{q} \quad Quit. Evaluates the current form and any other forms in the ordinary mode.
  \item \texttt{e form} \quad Eval. Evaluates the specified form in the ordinary mode and prints the resulting values. Then prompts again with the same current form.
  \item \texttt{?} \quad Help. Lists the stepper commands.
\end{itemize}

\section*{5.3 Errors}

\begin{flushright}
\texttt{*break-enable*}
\end{flushright}

\texttt{[Variable]}

This variable is used to determine whether to enter the break loop (see Section 5.4) when an error occurs. Even the function \texttt{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 error, 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
   noinline 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 popped away at the completion of the execution. An event envi-
ronment 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
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 exam-
ple. 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 error, error 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]

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

:v

[Abbreviated Break Loop Command]

: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]

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

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

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

:unhide-package package

: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 :hidè’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(current) ;;; Shows the current function.
:p(previous) ;;; To the previous function.
:n(next) ;;; To the next function.
:b(acktrace) ;;; Prints backtrace.
:help                  Help.
:quit                 Returns to top-level.
:resume               Returns to the caller of break-level.
:local                Shows the n-th local value on the stack.
:variables            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

>>> 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

>>> 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.

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**

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 described), 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 fast file (or a-file because of the actual filetype). The compiled program in a fast 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 1c or 1cl. These commands require the file name of the source file as their argument. Both 1c and 1cl simply add ".1sp" to the file name argument to obtain the full name of the source file.

% 1c 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).

\begin{tabular}{|c|}
\hline
\textbf{Temporary File} & \textbf{Contents} \\
\hline
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 \\
\hline
\end{tabular}

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 .lisp 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 fast 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.

\begin{tabular}{|c|c|}
\hline
\textbf{Output File} & \textbf{Filetype} \\
\hline
fast file & .o \\
c-file & .c \\
h-file & .h \\
data-file & .data \\
\hline
\end{tabular}

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 "F00.lsp" and the fast file is "F00.o" both in the current directory.

```
(compile-file 'foo.lish)
```

The source file is "F00.LISH" and the fast file is "F00.o".

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

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

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

If `definition` is not supplied, `name` should be the name of a not-yet-compiled function. In this case, `compiler` 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 `compiler` installs the compiled function as the function definition of `name` and returns `name`. If `name` is `nil`, then `compiler` 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 `compiler` actually creates a source file which contains the form designated by the arguments. Then `compiler` calls `compile-file` to get a fast file, which is then loaded into KCL. The source file and the fast file are given the names `gazonk.lsp` and `gazonk.fasl`, respectively. These files are not deleted automatically after the execution of `compiler`.

```
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, \texttt{thing} itself will be compiled as a top-level form.
In any case, \texttt{disassemble} does not install the compiled function.
\texttt{disassemble} returns no value.

No intermediate h-file is created if the keyword parameter \texttt{:h-file} is
\texttt{nil} or if \texttt{:h-file} is not supplied. Otherwise, an intermediate h-file
is created under the name specified by \texttt{:h-file}. Similarly, the inter-
mediate data-file is specified by the keyword parameter \texttt{:data-file}.

\begin{verbatim}
*eval-when-compile*
\end{verbatim}

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

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 proclaim returns t if the specified proclamation is still in effect. Otherwise, it returns nil. For example,

\[
(proclaim (special *x*)) ;; The variable *x* is nil
\]

\[
(proclamation (special *x*))
t
\]

\[
(defvar *y*) ;; Another way to proclaim a variable nil
\]

\[
(proclamation (special *y*))
t
\]

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,

\[
\text{the fixnum (foo)}
\]

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.

\[
special { variable-name }*
\]

[DeclarationSpecifier]

41
The interpreter and the compiler of KCL both treat special declarations exactly as described in the *Common Lisp Reference Manual.*

**type**

\[
\text{type} \{ \text{variable-name} \}* \quad \text{[DeclarationSpecifier]}
\]

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,

\[
>\text{(defun foo (x y)}
\quad (\text{declare (fixnum x) (character y)})
\quad (\text{setq x y)}
\quad ...))
\quad \text{foo}
\]

\[
>\text{(compile 'foo)}
\]

; (DEFUN FOO ...) is being compiled.

;; Warning: Type mismatches between X and Y.

See Section 7.3 for further information on type declarations.

**type**

\[
\text{type} \{ \text{variable-name} \}* \quad \text{[DeclarationSpecifier]}
\]

(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**

\[
\text{function-name argument-types . return-types} \quad \text{[DeclarationSpecifier]}
\]

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.

\[
\text{(defun foo ()}
\quad (\text{declare (function bar (character) fixnum))}
\quad (+ (bar (atcholi1)) (bar (atcholi2)))
\]

42
In this example, the function declaration specifies that the two functions atchol1 and atchol2 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,

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

(defun '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.

```lisp
(defun foo ()
  (declare (function bar (character character) fixnum))
  (+ (bar (atchol1)) (bar (atchol2) (atchol3) (atchol4))))
```

For this definition, the compiler assumes that the three functions atchol1, atchol2, and atchol3 will return fixnum objects. The return type of atchol4 is unknown at compile time.

The complete syntax of a function declaration is:

```lisp
(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 argument-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-name1 ... function-name0) is equivalent to n consecutive function declarations (function function-name1 . rest) ... (function function-name0 . 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

44
>(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 ... var_n) 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 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.

46
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 dereferencing 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.

### 7.3.1 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 do list may always be declared as object variables, since the do list 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.

7.3.2 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, zero?, plus?, minus?, =, /=, <, <=, >, >-, 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 aref, char, schar, and aref (see below). array
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).

7.3.3 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, and 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

51
built-in functions `<` and `>` 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` declaration and the local `fixnum` declaration are necessary to obtain the optimal code. The `function` declaration 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*.

```lisp
save filename
```

`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.

Example:

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

>(save "savefile")
%

% pwd
/usr/hagiya
% /usr/hagiya/savefile

>(plus 2 3)
5

>(bye)
Bye.
%
```

```lisp
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.

by &optional exit-code
by &optional exit-code

Terminates KCL and returns the exit-code to the parent process. exit-code must be an integer and its default value is 0.
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 defc 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 }*
  [
    [ [ &rest | &body ] pseudo-var ]
    [ &key { var | ( var | ( keyword pseudo-var ) [ initform [ pseudo-var ] ] ) }*[ allow-other-keys ]]
    [ &aux { var | ( pseudo-var [ initform ] )}* ]
  ]
)

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 ] ) } ]
  ]
)

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 and if 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 &rest 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 \texttt{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 fast-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 fast 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 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 objects and other C-types are primitive data types in the C language.

Clines \{string\}*

When the KCL compiler encounters a macro form (Clines string1 ...
stringgn), 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.

\texttt{defentry function parameter-list C-function} \quad \textbf{[Macro]}

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

\begin{verbatim}
(defentry tak (int int int) (int tak))
\end{verbatim}

The Lisp function \texttt{tak} defined by this \texttt{defentry} form requires three arguments. The arguments are converted to \texttt{int} values before they are passed to the C function. On return from the C function, the returned \texttt{int} value is converted to a Lisp integer (actually a \texttt{fixnum}) and this \texttt{fixnum} will be returned as the value of the Lisp function. (See below for type conversion between Lisp and the C language.) A \texttt{defentry} form is treated in the above way only when it appears as a top-level form of a Lisp source file. Otherwise, a \texttt{defentry} form expands to \texttt{nil}.

\texttt{defla name lambda-list \{declaration \mid doc-string\} \{form\}*} \quad \textbf{[Macro]}

When interpreted, \texttt{defla} is exactly the same as \texttt{defun}. That is, \texttt{(defla name lambda-list . body)} expands to \texttt{(defun name lambda-list . body)}. However, \texttt{defla} forms are completely ignored by the compiler; no C language code will be generated for \texttt{defla} forms. The primary use of \texttt{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. \texttt{defla} is short for \textit{DEFine Lisp Alternative}.

Suppose you have a Lisp source file whose contents are:

\begin{verbatim}
;;; C version of TAK.
(Clines

  %    int tak(x, y, z)
  %    int x, y, z;
\end{verbatim}
\% \{     \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 \texttt{tak} definition. Once this file has been compiled, and
when the generated fast file is loaded into KCL, a function call to
\texttt{tak} is actually the call to the C version of \texttt{tak}.

defCfun header n \{element\} \{Macro\}
defCfun defines a C language function which calls Lisp functions
and/or which handles Lisp objects. \textit{header} is a string consisting of
the C code for

1. the optional \textit{type-specifier} of the C function,
2. the \textit{function-declarator} of the C function, and
3. the \textit{type-decl-list} of the parameters to the C function.

(For the C language terminology, refer to \textit{The C Programming Lan-
guage} by Brian W. Kernighan and Dennis M. Ritchie.) The rest of
the C function definition, i.e., the \textit{function-statement}, is given by
\textit{elements}. Each \textit{element} may be a string, in which case the string is
treated in the same way as the arguments to the \texttt{Clines} macro. Or
else, the \textit{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 \textit{name}. As will be mentioned later, \textit{name}
may be quote, but \textit{name} may not be the name of any other special
form or a macro. The \textit{args} specify the arguments to the function and
the \textit{places} specify where the values should go. Thus the list-formed
\textit{element} could be regarded as something like the Lisp form:

61
(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 value (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.

```
(defun list2(x,y) 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.

```
(defun list2(x,y) 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 \{ C-type \}* )

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-exp | (C-type C-exp) \}

C-function-name:
C-exp:
\{ string | symbol \}

C-type:
\{ object | int | char | float | double \}
Chapter 11

The Editor

KCL/AOS is equipped with a screen editor FeCl2 (Full-screen Editor as a Common Lisp TOOL). FeCl2 is an EMACS-like editor with facilities for Lisp coding. FeCl2 is invoked from KCL by the function \texttt{ed} and the result of editing can be passed to KCL directly. For the details of FeCl2 refer to The FeCl2 Editor Reference Manual.

\texttt{ed} \texttt{&optional filename}  \hspace{1cm} \textit{[Function]}

\texttt{ed} invokes FeCl2 and sets the edit file of FeCl2 to \texttt{filename}. If the filetype of the file is not explicitly specified, then the \texttt{filename} is first merged into \\texttt{#".lsp".}

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

#### KCL Summary

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

<table>
<thead>
<tr>
<th>symbol</th>
<th>[kind]</th>
<th>remark</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>*</code></td>
<td>[Function]</td>
<td></td>
</tr>
<tr>
<td><code>*</code></td>
<td>[Variable]</td>
<td></td>
</tr>
<tr>
<td><code>**</code></td>
<td>[Variable]</td>
<td></td>
</tr>
<tr>
<td><code>***</code></td>
<td>[Variable]</td>
<td></td>
</tr>
<tr>
<td><code>+</code></td>
<td>[Function]</td>
<td></td>
</tr>
<tr>
<td><code>+</code></td>
<td>[Variable]</td>
<td></td>
</tr>
<tr>
<td><code>++</code></td>
<td>[Variable]</td>
<td></td>
</tr>
<tr>
<td><code>+++</code></td>
<td>[Variable]</td>
<td></td>
</tr>
<tr>
<td><code>-</code></td>
<td>[Function]</td>
<td></td>
</tr>
<tr>
<td><code>/</code></td>
<td>[Variable]</td>
<td></td>
</tr>
<tr>
<td><code>/</code></td>
<td>[Function]</td>
<td></td>
</tr>
<tr>
<td><code>//</code></td>
<td>[Variable]</td>
<td></td>
</tr>
<tr>
<td><code>///</code></td>
<td>[Variable]</td>
<td></td>
</tr>
<tr>
<td><code>/=</code></td>
<td>[Function]</td>
<td></td>
</tr>
<tr>
<td><code>1+</code></td>
<td>[Function]</td>
<td></td>
</tr>
<tr>
<td><code>1-</code></td>
<td>[Function]</td>
<td></td>
</tr>
</tbody>
</table>

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]
<= [Function]
= [Function]
> [Function]
>= [Function]
:abort [Keyword]
abs [Function]
acos [Function]
acosh [Function]
adjoin [Function]
adjust-array [Function]
:adjustable [Keyword]
adjustable-array-p [Function]
allocate [Function]
allocate-contiguous-pages [Function]
allocated-contiguous-pages [Function]
allocated-pages [Function]
allocated-relocatable-pages [Function]
allocate-relocatable-pages [Function]
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]
: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]
bool  [Function]
bool-1 [Constant]
bool-2 [Constant]
bool-and [Constant]
bool-andc1 [Constant]
bool-andc2 [Constant]
bool-clr [Constant]
bool-c1 [Constant]
bool-c2 [Constant]
bool-eqv [Constant]
bool-ior [Constant]
bool-nand [Constant]
boolean [Constant]
boolean-orc1 [Constant]
boolean-orc2 [Constant]
boolean-set [Constant]
boolean-xor [Constant]
both-case-p [Function]
boundp [Function]
*break-on-warnings* [Variable]
butlast [Function]
by [Function] Added. See pages 2 and 54.
bye [Function] Added. See pages 2 and 54.
byte [Function]
byte-position [Function]
byte-size [Function]
calpha [Function]
caaddr [Function]
caar [Function]
caadar [Function]
caaddr [Function]
caadr [Function]
caar [Function]
caadaar [Function]
caadadr [Function]
cadar [Function]
caddar [Function]
caddadr [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]

69
cdadr [Function]
cdar [Function]
cddar [Function]
cddadr [Function]
cddar [Function]
cdddar [Function]
cdddr [Function]
cddr [Function]
cdr [Function]
ceiling [Function]
error [Function]
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. See page 17.
close [Function] Different. See page 17.
clrhash [Function]
code-char [Function]
coerce [Function] See page 64.
commonp [Function]
compile [Function] See page 38.
compiled-function-p [Function]
compiler-let [Special]
complex [Function]
complexp [Function]
:conc-name [Keyword]
concatenate [Function]
cond [Special, Macro]
conjugate [Function]
cons [Function]
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]
<table>
<thead>
<tr>
<th><strong>defun</strong></th>
<th>[Special, Macro] Added. See page 60.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>defvar</strong></td>
<td>[Macro] See page 40.</td>
</tr>
<tr>
<td><strong>delete</strong></td>
<td>[Function]</td>
</tr>
<tr>
<td><strong>delete-duplicates</strong></td>
<td>[Function]</td>
</tr>
<tr>
<td><strong>delete-file</strong></td>
<td>[Function]</td>
</tr>
<tr>
<td><strong>delete-if</strong></td>
<td>[Function]</td>
</tr>
<tr>
<td><strong>delete-if-not</strong></td>
<td>[Function]</td>
</tr>
<tr>
<td><strong>denominator</strong></td>
<td>[Function]</td>
</tr>
<tr>
<td><strong>deposit-field</strong></td>
<td>[Function]</td>
</tr>
<tr>
<td><strong>describe</strong></td>
<td>[Function] See page 34.</td>
</tr>
<tr>
<td><strong>:device</strong></td>
<td>[Keyword]</td>
</tr>
<tr>
<td><strong>digit-char</strong></td>
<td>[Function]</td>
</tr>
<tr>
<td><strong>digit-char-p</strong></td>
<td>[Function]</td>
</tr>
<tr>
<td><strong>:direction</strong></td>
<td>[Keyword]</td>
</tr>
<tr>
<td><strong>directory</strong></td>
<td>[Function]</td>
</tr>
<tr>
<td><strong>:directory</strong></td>
<td>[Keyword]</td>
</tr>
<tr>
<td><strong>directory-namestring</strong></td>
<td>[Function]</td>
</tr>
<tr>
<td><strong>disassemble</strong></td>
<td>[Function] Different. See page 38.</td>
</tr>
<tr>
<td><strong>:displaced-index-offset</strong></td>
<td>[Keyword]</td>
</tr>
<tr>
<td><strong>:displaced-to</strong></td>
<td>[Keyword]</td>
</tr>
<tr>
<td><strong>do</strong></td>
<td>[Special, Macro]</td>
</tr>
<tr>
<td><strong>do</strong></td>
<td>[Special, Macro]</td>
</tr>
</tbody>
</table>
do-all-symbols
 do-external-symbols
 do-symbols
 documentation
 dolist
 dotimes
 double-float-epsilon
 double-float-negative-epsilon
 dpb
 dribble
 ecase
 ed
 eighth
 :element-type
 elt
 encode-universal-time
 :end
 :end1
 :end2
 endp
 enough-namestring
 &environment

 eq
 eql
 equal
 equalp
 error
 :error
 *error-output*
 :escape
 etypecase
 eval
 evalhook
 *evalhook*
 eval-when
 *eval-when-compile*
 evenp
 every
 exp
 export
 expit
 :external

 [Macro]
 [Macro]
 [Macro]
 [Function]
 [Special, Macro]
 [Special, Macro]
 [Constant]
 [Constant]
 [Function]
 [Function]
 [Function]
 [Function]
 [Function]
 [Function]
 [Function]
 [Keyword]
 [Keyword]
 [Keyword]
 [Keyword]
 [Function]
 [Function]
 [Function]
 [Function]
 [Function]
 [Function]
 [Function]
 [Function]
 [Function]
 [Function]
 [Function]
 [Function]
 [Keyword]

 See page 56.

 See page 17.

 Different. See page 65.

 See page 29.

 See pages 33 and 33.

 See page 36.

 Added. See pages 36 and 39.
fboundp [Function]
flooring [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]
fllet [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.
        See page 44.
funcall [Function]
function [Special] Also Declaration Specifier.
        See page 42.
functionp [Function]

74
:functions

[Keyword] Break Loop Command.
See page 32.

gbc

gcd

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-self-method

[Function]
get-self-method-multiple-value

[Function]
get-universal-time

[Function]

tf

[Function]
gethash

[Function]
go

[Special] See pages 30 and 32.

graphic-char-p

[Function]

dash-table-count

[Function]
dash-table-p

[Function]

: h

See page 31.

: help

[Keyword] Break Loop Command.
See page 31.

: h-file


: hide

[Keyword] Break Loop Command.
See page 32.

: hide-package

[Keyword] Break Loop Command.
See page 33.

: host

[Keyword]

host-namestring

[Function]

identity

[Function]

if

[Special]

:if-does-not-exist

[Keyword]

:if-exists

[Keyword]

ignore

[Symbol] Declaration Specifier.
See page 45.

*ignore-maximum-pages*

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. See page 44.
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]
isqrt  [Function]
:junk-allowed  [Keyword]
:key  [Keyword]
keywordp  [Function]
labels  [Special]
lambda-list-keywords  [Constant]
lambda-parameters-limit  [Constant]
last  [Function]
1cm  [Function]
1db  [Function]
1db-test  [Function]
1diff  [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]
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>least-positive-long-float</code></td>
<td>[Constant]</td>
</tr>
<tr>
<td><code>least-positive-short-float</code></td>
<td>[Constant]</td>
</tr>
<tr>
<td><code>least-positive-single-float</code></td>
<td>[Constant]</td>
</tr>
<tr>
<td><code>length</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>:length</code></td>
<td>[Keyword]</td>
</tr>
<tr>
<td><code>let</code></td>
<td>[Special]</td>
</tr>
<tr>
<td><code>let*</code></td>
<td>[Special]</td>
</tr>
<tr>
<td><code>:level</code></td>
<td>[Keyword]</td>
</tr>
<tr>
<td><code>lisp-implementation-type</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>lisp-implementation-version</code></td>
<td>[Function]</td>
</tr>
<tr>
<td>See page 2.</td>
<td></td>
</tr>
<tr>
<td><code>list</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>list*</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>list-all-packages</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>list-length</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>listen</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>Different. See page 17.</code></td>
<td></td>
</tr>
<tr>
<td><code>listp</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>load</code></td>
<td>[Function]</td>
</tr>
<tr>
<td>See pages 17 and 36.</td>
<td></td>
</tr>
<tr>
<td><code>*load-verbose*</code></td>
<td>[Variable]</td>
</tr>
<tr>
<td><code>:local</code></td>
<td>[Keyword]</td>
</tr>
<tr>
<td>Break Loop Command.</td>
<td></td>
</tr>
<tr>
<td>See page 32.</td>
<td></td>
</tr>
<tr>
<td><code>locally</code></td>
<td>[Special, Macro]</td>
</tr>
<tr>
<td><code>log</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>logand</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>logandc1</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>logandc2</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>logbitp</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>logcount</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>logeqv</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>loglor</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>lognand</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>lognor</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>lognot</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>logorc1</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>logorc2</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>logtest</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>logxor</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>long-float-epsilon</code></td>
<td>[Constant]</td>
</tr>
<tr>
<td><code>long-float-negative-epsilon</code></td>
<td>[Constant]</td>
</tr>
<tr>
<td><code>long-site-name</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>loop</code></td>
<td>[Special, Macro]</td>
</tr>
<tr>
<td><code>lower-case-p</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>machine-instance</code></td>
<td>[Function]</td>
</tr>
<tr>
<td><code>machine-type</code></td>
<td>[Function]</td>
</tr>
<tr>
<td>Function</td>
<td>Function</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>machine-version</td>
<td>macro-function</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>make-array</td>
<td>make-broadcast-stream</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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]

:name [Keyword]
name-char [Function]
:name [Keyword]
namestring [Function]
nbutlast [Function]
nconc [Function]
:new-version [Keyword]

nil [Constant]
nintersection [Function]
ninth [Function]
not [Function]
notany [Function]
notevery [Function]
notinline [Symbol] Declaration Specifier. See pages 27, 30, and 44.
nreconc [Function]
nreverse [Function]
nset-difference [Function]
nset-exclusive-or [Function]
string-case [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]

oddp  [Function]
open  [Function] Different. See page 17.
optimize  [Symbol] Declaration Specifier. See page 45.

or  [Special, Macro]
output-stream-p  [Function]
:overwrite  [Keyword]
*package*  [Variable] See page 2.
package-name  [Function]
package-nicknames  [Function]
package-shadowing-symbols  [Function]
package-use-list  [Function]
package-used-by-list  [Function]
packagep  [Function]
pairlis  [Function]
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] See page 8.
plusp  [Function]
pop  [Macro]
position  [Function]
position-if  [Function]
positional-not  [Function]
pprint  [Function]
    See page 31.
:predicate  [Keyword]
:preserve-whitespace  [Keyword]
:pretty  [Keyword]
:previous  [Keyword] Break Loop Command.
    See page 31.
print  [Function]
print-to-string  [Function]
princ  [Function] See page 17.
princ-to-string  [Function]
print  [Function]
*print-base*  [Variable]
*print-case*  [Variable]
*print-circle*  [Variable]
*print-escape*  [Variable]
:print-function  [Keyword]
*print-gensym*  [Variable]
*print-length*  [Variable] See page 28.
*print-level*  [Variable] See page 28.
*print-pretty*  [Variable]
*print-radix*  [Variable]
:probe  [Keyword]
prog  [Special, Macro]
prog*  [Special, Macro]
progl  [Special, Macro]
prog2  [Special, Macro]

81
progn
progv
provide
psetf
psetq
push
pushnew
:q
*query-io*
:quit
quote
:r
:radix
random
*random-state*
random-state-p
rassoc
rassoc-if
rassoc-if-not
rational
rationalize
rationalp
read
*read-base*
read-byte
read-char
read-char-no-hang
*read-default-float-format*
read-delimited-list
read-from-string
read-line
:read-only
read-preserving-whitespace
*read-suppress*
*readtable*
readtablep
realpart
reduce
:rehash-size
:rehash-threshold
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]
                     See page 31.
rest [Function]
return [Special, Macro]
return-from [Special] See page 32.
revappend [Function]
reverse [Function]
rotatef [Macro]
round [Function]
rplaca [Function]
rplacd [Function]
safety [Symbol] Optimize Quality.
                     See pages 30, 45, and 57.
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
seventh
shadow
shadowing-import
shift
short-float-epsilon
short-float-negative-epsilon
short-site-name
signum
simple-bit-vector-p
simple-string-p
simple-vector-p
sin
single-float-epsilon
single-float-negative-epsilon
sinh
sixth
:size
:sleep
software-type
software-version
some
sort
space
special

special-form-p
speed
sqrt
stable-sort
standard-char-p
*standard-input*
*standard-output*
:start
:start1
:start2
:static
step
:stream
stream-element-type
streamamp
string
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]
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]
t [Constant]
See page 32.
tagbody  [Special] See page 32.
tailp  [Function]
tan  [Function]
tanh  [Function]
tenth  [Function]
*terminal-is*  [Variable] Different. See pages 16 and 18.
terpri  [Function]
test  [Keyword]
test-not  [Keyword]
the  [Special] See pages 41 and 50.
third  [Function]
throw  [Special]
time  [Macro]
trace  [Macro] See page 27.
*trace-output*  [Variable]
tree-equal  [Function]
truename  [Function]
truncate  [Function]
type  [Symbol] Declaration Specifier. See pages 42 and 47.
:type  [Keyword]
type-of  [Function]
typcase  [Macro]
typep  [Function] See page 41.
unexport  [Function]
unintern  [Function]
union  [Function]

unless  [Special, Macro]
unread-char  [Function]
untrace  [Macro] See page 27.
unuse-package  [Function]
unwrap-protect  [Special]
upper-case-p  [Function]
use-package  [Function]
user-homedir-pathname  [Function]
values [Function] See pages 44 and 60.
values-list [Function]

vector [Function]
vector-pop [Function]
vector-push [Function]
vector-push-extend [Function]
vectorp [Function]
:version [Keyword]
warn [Function]
when [Special, Macro]
&whole [Symbol] Defmacro-lambda Keyword. See page 56.


with-input-from-string [Macro]
with-open-file [Macro]
with-open-stream [Macro]
with-output-to-string [Macro]
write [Function]
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]