Lisp Programming Interview Questions And Answers Guide.

Global Guideline.

https://globalguideline.com/

Lisp Programming Job Interview Preparation Guide.

Question # 1

Explain What is the "minimal" set of primitives needed for a Lisp interpreter?

Answer:-

Many Lisp functions can be defined in terms of other Lisp functions.
For example, CAAR can be defined in terms of CAR as
(defun caar (list) (car (car list)))
It is then natural to ask whether there is a "minimal" or smallest set
of primitives necessary to implement the language.
There is no single "best" minimal set of primitives; it all depends on
the implementation. For example, even something as basic as numbers
need not be primitive, and can be represented as lists. One possible
set of primitives might include CAR, CDR, and CONS for manipulation of
S-expressions, READ and PRINT for the input/output of S-expressions
and APPLY and EVAL for the guts of an interpreter. But then you might
want to add LAMBDA for functions, EQ for equality, COND for
conditionals, SET for assignment, and DEFUN for definitions. QUOTE
might come in handy as well. If you add more specialized datatypes,
such as integers, floats, arrays, characters, and structures, you'll
need to add primitives to construct and access each.
AWKLisp is a Lisp interpreter written in awk, available by anonymous
ftp from ftp.cs.cmu.edu:/user/ai/lang/lisp/impl/awk/. It has thirteen
built-in functions: CAR, CDR, CONS, EQ, ATOM, SET, EVAL, ERROR, QUOTE,
COND, AND, OR, LIST.
A more practical notion of a "minimal" set of primitives might be to
look at the implementation of Scheme. While many Scheme functions can
be derived from others, the language is much smaller than Common Lisp.
See Dybvig's PhD thesis,
R. Kent Dybvig, "Three Implementation Models for Scheme", Department
of Computer Science Technical Report #87-011, University of North
Carolina at Chapel Hill, Chapel Hill, North Carolina, April 1987.
for a justification of a particularly practical minimal set of
primitives for Scheme.
In a language like Common Lisp, however, there are a lot of low-level
primitive functions that cannot be written in terms of the others,
such as GET-UNIVERSAL-TIME, READ-CHAR, WRITE-CHAR, OPEN, and CLOSE,
for starters. Moreover, real Common Lisp implementations are often
built upon primitives that aren't part of the language, per se, and
certainly not intended to be user-accessible, such as SYS:%POINTER-REF.
Beside the references listed in [1-4], some other relevant references
include:
McCarthy, John, "Recursive Functions of Symbolic Expressions and
their Computation by Machine, Part I", CACM 3(4):185-195, April 1960.
[Defines five elementary functions on s-expressions.]
McCarthy, John, "A Micro-Manual for Lisp -- not the whole Truth",
ACM SIGPLAN Notices, 13(8):215-216, August 1978.
[Defines the Lisp programming language in 10 rules and gives
a small interpreter (eval) written in this Lisp.]
McCarthy, John, et al., "LISP 1.5 Programmer's Manual", 2nd edition,
MIT Press, 1965, ISBN 0-262-13011-4 (paperback).
[Gives five basic functions, CAR, CDR, CONS, EQ, and ATOM.
Using composition, conditional expressions (COND), and
recursion, LAMBDA, and QUOTE, these basic functions may be used
to construct the entire class of computable functions of
S-expressions. Gives the functions EVAL and APPLY in
M-expression syntax.]

Read More Answers.

Question # 2

How to pass commands to LG3?

Answer:-

By selecting general operations from the tools menu. You can think of the tools as building blocks - each tool corresponding to several lines of LISP code. The tools you use and the order in which you select them defines what your program does.

Read More Answers.

Question # 3

Lisp books, introductions, documentation, periodicals, journals, and conference proceedings!

Answer:-

There are several good Lisp introductions and tutorials:
1. David S. Touretzky
"Common Lisp: A Gentle Introduction to Symbolic Computation"
Benjamin/Cummings Publishers, Redwood City, CA, 1990. 592 pages.
ISBN 0-8053-0492-4 ($42.95).
Perhaps the best tutorial introduction to the language. It has
clear and correct explanations, and covers some fairly advanced
topics. The book is an updated Common Lisp version of the 1984
edition published by Harper and Row Publishers.
Three free Lisp educational tools which were used in the book --
Evaltrace, DTRACE and SDRAW -- are available by anonymous ftp from
b.gp.cs.cmu.edu:/usr/dst/public/lisp/
b.gp.cs.cmu.edu:/usr/dst/public/evaltrace/
Evaltrace is a graphical notation for explaining how evaluation
works and is described in "Visualizing Evaluation in
Applicative Languages" by David S. Touretzky and Peter Lee,
CACM 45-59, October 1992. DTRACE is a "detailed trace" which
provides more information than the tracing tools provided with
most Common Lisp implementations. SDRAW is a read-eval-draw
loop that evaluates Lisp expressions and draws the result as a
cons cell diagram (for both X11 and ascii terminals). Also
available is PPMX, a tool for pretty printing macro expansions.
2. Robert Wilensky.
"Common LISPcraft"
W. W. Norton, New York, 1986. 500 pages. ISBN 0-393-95544-3.
3. Wade L. Hennessey.
"Common Lisp"
McGraw-Hill, New York, 1989. 395 pages. ISBN 0-07-028177-7, $26.95.
Fairly good, but jumps back and forth from the simple to the
complex rather quickly, with no clear progression in difficulty.
4. Laurent Siklossy.
"Let's Talk LISP"
Prentice-Hall, NJ, 1976. 237 pages, ISBN 0-13-53276-2-8.
Good introduction, but quite out of date.
5. Stuart C. Shapiro.
"Common Lisp: An Interactive Approach"
Computer Science Press/W.H. Freeman, New York, 1992.
358 pages, ISBN 0-7167-8218-9.
The errata for the book may be obtained by anonymous ftp from
ftp.cs.buffalo.edu:/users/shapiro/clerrata.ps
Other introductions to Lisp include:
1. A. A. Berk.
"LISP, The Language of Artificial Intelligence"
Van Nostrand Reinhold, 1985. 160 pages, ISBN 0-44-22097-4-6.
2. Paul Y. Gloess.
"An Alfred handy guide to Understanding LISP"
Alfred Publishers (Sherman Oaks, CA), 1982.
64 pages, ISBN 0-88-28421-9-6, $2.95.
3. Ward D. Maurer.
"The Programmer's Introduction to LISP"
American Elsevier, New York, 1972. 112 pages, ISBN 0-44-41957-2-6.
4. Hank Bromley and Richard Lamson.
"LISP Lore: A Guide to Programming the LISP Machine", 2nd edition
Kluwer Academic, Boston, 1987. 337 pages, ISBN 0-89-83822-8-9, $49.95.
5. Sharam Hekmatpour.
"Introduction to LISP and Symbol Manipulation"
Prentice Hall, New York, 1989. 303 pages, ISBN 0-13-53749-0-1, $40.
6. Deborah G. Tatar
"A programmer's guide to Common Lisp"
Digital Press, 1987. 327 pages. ISBN 0-932376-87-8.
Good introduction on Common Lisp for programmers familiar
with other programming languages, such as FORTRAN, PASCAL, or C.
7. Timothy Koschmann
"The Common Lisp Companion"
John Wiley & Sons, 1990. 459 pages, ISBN 0-471-503-8-8.
Targeted for those with some programming experience who wish to
learn draft-ANSI Common Lisp, including CLOS and the CL condition
system. Examples progress incrementally from simple numerical
calculation all the way to a logic-programming extension to CL.
More advanced introductions to Lisp and its use in Artificial

Intelligence include:
1. Peter Norvig.
"Paradigms of AI Programming: Case Studies in Common Lisp"
Morgan Kaufmann, 1992. 946 pages. ISBN 1-55860-191-0 ($49.95).
Provides an in-depth exposition of advanced AI programming techniques and includes large-scale detailed examples. The book is the most advanced AI/Common-Lisp programming text and reference currently available, and hence is not for the complete novice. It focuses on the programming techniques necessary for building large AI systems, including object-oriented programming, and has a strong performance orientation.
The text is marked by its use of "non-toy" examples to illustrate the techniques. All of the examples are written in Common Lisp, and copies of the source code are available by anonymous ftp from unix.sri.com:/pub/norvig and on disk in Macintosh or DOS format from the publisher. Some of the techniques described include rule-based pattern matching (GPS, Eliza, a subset of Macsyma, the Emycin expert system shell), constraint propagation and backtracking (Waltz line-labelling), alpha-beta search (Othello), natural language processing (top-down, bottom-up and chart parsing), logic-programming (unification and Prolog), interpreters and compilers for Scheme, and object-oriented programming (CLOS).
The examples are also used to illustrate good programming style and efficiency. There is a guide to trouble-shooting and debugging Lisp programs, a style guide, and a discussion of portability problems.
Some of the efficiency techniques described include memoization, data indexing, compilation, delaying computation, proper use of declarations, avoiding garbage collection, and choosing and using the correct data structure.
The book also serves as an advanced introduction to Common Lisp, with sections on the Loop macro, CLOS and sequences, and some coverage of error handling, series, and the package facility.
2. Eugene Charniak, Christopher K. Riesbeck, Drew V. McDermott and James R. Meehan.
"Artificial Intelligence Programming", 2nd edition.
Lawrence Erlbaum Associates (Hillsdale, NJ), 1987.
533 pages, ISBN 0-89-85960-9-2, $29.95.
Provides many nice code fragments, all of which are written in Common Lisp. The first half of the book covers topics like macros, the reader, data structures, control structures, and defstructs. The second half of the book describes programming techniques specific to AI, such as discrimination nets, production systems, deductive database retrieval, logic programming, and truth maintenance.
3. Patrick H. Winston and Berthold K. P. Horn.
"LISP", 3rd edition.
Addison-Wesley (Reading, MA), 1989. 611 pages. ISBN 0-201-08319-1
Covers the basic concepts of the language, but also gives a lot of detail about programming AI topics such as rule-based expert systems, forward chaining, interpreting transition trees, compiling transition trees, object oriented programming, and finding patterns in images. Not a tutorial. Has many good examples. Source code for the examples is available by anonymous ftp from ftp.ai.mit.edu:/pub/lisp3/. (The code runs in Lucid, Allegro, KCL, GCLisp, MCL, Symbolics Genera. Send mail with subject line "help" to ai3@ai.mit.edu for more information.)
4. John R. Anderson, Albert T. Corbett, and Brian J. Reiser.
"Essential LISP"
Addison-Wesley (Reading, MA), 1987.
352 pages, ISBN 0-20-11114-8-9, $23.95.
Concentrates on how to use Lisp with iteration and recursion.
5. Robert D. Cameron and Anthony H. Dixon
"Symbolic Computing with Lisp"
Prentice-Hall, 1992, 326 pages. ISBN 0-13-877846-9.
The book is intended primarily as a third-year computer science text. In terms of programming techniques, it emphasizes recursion and induction, data abstraction, grammar-based definition of Lisp data structures and functional programming style. It uses two Lisp languages:
(1) a purely functional subset of Lisp called Small Lisp and
(2) Common Lisp.
An MS-DOS interpreter for Small Lisp (including source) is provided with the book. It considers applications of Lisp to formal symbolic data domains: algebraic expressions, logical formulas, grammars and programming languages.
6. Tony Hasemer and John Domingue.
"Common Lisp Programming for Artificial Intelligence"
Addison-Wesley, Reading, MA, 1989. 444 pages, ISBN 0-20-11757-9-7.
This book presents an introduction to Artificial Intelligence with an emphasis on the role of knowledge representation. Three chapters focus on object-oriented programming, including the construction and use of a subset of CLOS.
The authors' research into the problems faced by novice Lisp

users influenced the content and style of the book. (The authors
are members of the Human Cognition Research Laboratory at the
Open University in the United Kingdom.) The book employs a
tutorial approach, especially in areas that students often find
difficult, such as recursion. Early and progressive treatment of
the evaluator promotes understanding of program execution.
Hands-on exercises are used to reinforce basic concepts.
The book assumes no prior knowledge of Lisp or AI and is a
suitable textbook for students in Cognitive Science, Computer
Science and other disciplines taking courses in Lisp or AI
programming as well as being invaluable for professional
programmers who are learning Lisp for developing AI applications.
7. Steven Tanimoto
"The Elements of Artificial Intelligence Using Common Lisp", 2nd edition
Computer Science Press, New York, 1995.
562 pages, ISBN 0-71-67826-9-3, (ISBN 0-71-67823-0-8, 1990, $48).
8. Patrick R. Harrison
"Common Lisp and Artificial Intelligence"
Prentice Hall, Englewood Clifs, NJ, 1990.
244 pages, ISBN 0-13-1552430, $22.50.
9. Paul Graham
"On Lisp: Advanced Techniques for Common Lisp"
Prentice Hall, Englewood Clifs, NJ, 1994. 413 pages, ISBN 0-13-030552-9.
Emphasizes a bottom-up style of writing programs, which he
claims is natural in Lisp and has advantages over the
traditional way of writing programs in C and Pascal.
Also has in-depth sections on writing macros with several
nice examples. Source code is available by anonymous ftp from
ftp.das.harvard.edu:/pub/onlisp/
as a single 56kb file.
10. John A. Moyne
"Lisp: A first language for computing"
Van Nostrand Reinhold, New York, 1991. 278 pages, ISBN 0442004265.
General Lisp reference books include:
1. ANSI/X3J13
Programming Language Common Lisp (ANSI/X3.226-1994)
American National Standards Institute
11 West 42nd Street, New York, NY 10036.
http://www.ansi.org/
2. Kent M. Pitman
Common Lisp HyperSpec (TM)
Harlequin, Inc., 1996.
This is an HTML-only document available via the web.
Available for browsing from
http://www.harlequin.com/books/HyperSpec/FrontMatter/
Available free for download (subject to some legal restrictions) from
http://www.harlequin.com/books/HyperSpec/
Includes text from ANSI/X3.226-1994 and other design rationales.
3. Guy L. Steele
"Common Lisp: The Language" [CLtL1]
Digital Press, 1984. 465 pages. ISBN 0-932376-41-X.
4. Guy L. Steele
"Common Lisp: The Language, 2nd Edition" [CLtL2]
Digital Press, 1990. 1029 pages, ISBN 1-55558-041-6 paperbound ($39.95).
[Butterworth-Heinemann, the owners of Digital Press, have made
the LaTeX sources to this book available by anonymous FTP from
cambridge.apple.com:/pub/CLTL/
A copy of the distribution is also available from
ftp.cs.cmu.edu:/user/ai/lang/lisp/doc/cltl/
The paperbound version of the book is, of course, available at
fine bookstores, or contact them directly at Digital Press,
225 Wildwood Street, Woburn, MA 01801, call 800-366-2665
(617-928-2527), or fax 800-446-6520 (617-933-6333). A copy of
the Digital Press book catalog is available from the same FTP location.]
A html version, produced using latex2html on the latex sources,
is accessible via the URL:
http://www.cs.cmu.edu/Web/Groups/AI/html/cltl/cltl2.html
5. Franz Inc.
"Common Lisp: The Reference"
Addison-Wesley, Reading, MA 1988. ISBN 0-201-11458-5
Entries on Lisp (CLtL1) functions in alphabetical order.

Read More Answers.

Question # 4

What if we get interrupted?

Answer:-

You are free to come and go from the LISP Generator and do whatever you want in AutoCAD while you are in the middle of creating a program. There are helpful tools in case you forget things like the names of variables you defined.

Read More Answers.

Question # 5

How complex can we get?

Answer:-

If you want, extremely complex. The LISP Generator utilizes nearly the entire AutoLISP language. There's no limit to how large your programs can be, and no limit to how complex they can be either. Even though most programs can be written in a straightforward or "linear" manner, you have the option to write highly embedded code. You can nest multiple IF and LOOP statements within each other. Also, you can always feed complex operations or math equations as direct input to larger operations, which themselves could be embedded or "nested" in other operations, and so on. So, the sky's the limit.

Read More Answers.

Question # 6

How to improve a Lisp programming style and coding efficiency?

Answer:-

There are several books about Lisp programming style, including:
1. Molly M. Miller and Eric Benson
"Lisp Style and Design"
Digital Press, 1990. 214 pages, ISBN 1-55558-044-0, $26.95.
How to write large Lisp programs and improve Lisp programming
style. Uses the development of Lucid CL as an example.
2. Robin Jones, Clive Maynard, and Ian Stewart.
"The Art of Lisp Programming"
Springer-Verlag, 1989. 169 pages, ISBN 0-387-19568-8 ($33).
3. W. Richard Stark.
"LISP, Lore, and Logic: An Algebraic View of LISP
Programming, Foundations, and Applications"
Springer-Verlag, 1990. 278 pages. ISBN 0-387-97072-X paper ($42).
Self-modifying code, self-reproducing programs, etc.
4. CMU CL User's Manual, Chapter 7, (talks about writing
efficient code). It is available by anonymous ftp from any CMU CS
machine (e.g., ftp.cs.cmu.edu [128.2.206.173]) as the file
/afs/cs.cmu.edu/project/clisp/docs/cmu-user/cmu-user.ps
[when getting this file by anonymous ftp, one must cd to
the directory in one atomic operation, as some of the superior
directories on the path are protected from access by anonymous ftp.]
5. See also Norvig's book, SICP (Abelson & Sussman), SAP
(Springer and Friedman).
6. Hallvard Tretteberg's Lisp Style Guide is available by anonymous
ftp in ftp.think.com:/public/think/lisp/style-guide.text. There is
a fair bit of overlap between Hallvard's style guide and the notes
below and in part 3 of this FAQ.
7. Rajeev Sangal
"Programming Paradigms in Lisp"
McGraw-Hill, 1991. ISBN 0-07-054666-5.
8. Rodney A. Brooks.
"Programming in Common Lisp"
John Wiley & Sons, New York, 1985. 303 pages. ISBN 0-471-81888-7.
Chapter 5 discusses Lisp programming style.
Here are some general suggestions/notes about improving Lisp
programming style, readability, correctness and efficiency:
General Programming Style Rules:
- Write short functions, where each function provides a single,
well-defined operation. Small functions are easier to
read, write, test, debug, and understand.
- Use descriptive variable and function names. If it isn't clear
from the name of a function or variable what its purpose is,
document it with a documentation string and a comment. In fact,
even if the purpose is evident from the name, it is still worth
documenting your code.
- Don't write Pascal (or C) code in Lisp. Use the appropriate
predefined functions -- look in the index to CLtL2, or use the
APROPOS and DESCRIBE functions. Don't put a close parenthesis
on a line by itself -- this can really irritate programmers
who grew up on Lisp. Lisp-oriented text editors include tools
for ensuring balanced parentheses and for moving across
pairs of balanced parentheses. You don't need to stick
comments on close parentheses to mark which expression they close.
- Use proper indentation -- you should be able to understand
the structure of your definitions without noticing the parentheses.
In general, the way one indents a form is controlled by the
first symbol of the form. In DEFUNs, for example, one puts the
symbol DEFUN, the function name, and the argument list all on
the same line. If the argument list is too long, one can break
it at one of the lambda keywords. Following the argument list,
one inserts a carriage return and lists the expressions in the
body of the definition, with each form starting on its own
line indented three spaces relative to the open parenthesis of
the parent (in this case the DEFUN). This general style -- of
putting all the significant elements of a form on a single
line, followed by a carriage return and the indented body --
holds for many Lisp constructs. There are, of course, variations,

such as keeping the first clause on the same line as the COND
or CASE symbol, and the rules are relaxed in different ways to
keep line lengths to a manageable size. If you find yourself having
trouble fitting everything in even with line breaking and
relaxing the rules, either your function names are too long or your
code isn't very modular. You should perceive this as a signal that
you need to break up your big definitions into smaller chunks, each
with a clearly defined purpose, and possibly replace long function
names with concise but apt shorter ones.
- Use whitespace appropriately. Use whitespace to separate
semantically distinct code segments, but don't use too much
whitespace. For example,
GOOD:

```
(defun foo (x y)
(let ((z (+ x y 10)))
(* z z)))
```

BAD:

```
(defun foo(x y)(let((z(+ x y 10)))(* z z)))
(defun foo ( x y )
(let ( ( z (+ x y 10) ) )
( * z z )
)
)
```

Although the Lisp reader and compiler don't care which you
use, most experienced Lisp programmers find the first example
much easier to read than the last two.
- Don't use line lengths greater than 80 characters. People who
write code using Zmacs on Symbolics Lisp Machines are notoriously
guilty of violating this rule, because the CPT6 font allows
one to squeeze a tremendous amount of code on the display,
especially if one spreads the code out horizontally. This
makes it more difficult to read when printed out or read on
an 80x24 xterm window. In fact, use a line length of 72 characters
because it leaves a strip of white space at the edge of the window.
The following functions often abused or misunderstood by novices.
Think twice before using any of these functions.
- EVAL. Novices almost always misuse EVAL. When experts use
EVAL, they often would be better off using APPLY, FUNCALL, or
SYMBOL-VALUE. Use of EVAL when defining a macro should set off
a warning bell -- macro definitions are already evaluated
during expansion. See also the answer to question 3-12.
The general rule of thumb about EVAL is: if you think you need
to use EVAL, you're probably wrong.
- PROGV. PROGV binds dynamic variables and is often misused in
conjunction with EVAL, which uses the dynamic environment.
In general, avoid unnecessary use of special variables.
PROGV is mainly for writing interpreters for languages embedded
in Lisp. If you want to bind a list of values to a list of
lexical variables, use
(MULTIPLE-VALUE-BIND (..) (VALUES-LIST ..) ..)
or
(MULTIPLE-VALUE-SETQ (..) (VALUES-LIST ..))
instead. Most decent compilers can optimize this expression.
However, use of this idiom is not to be encouraged unless absolutely
necessary.
- CATCH and THROW. Often a named BLOCK and RETURN-FROM are
more appropriate. Use UNWIND-PROTECT when necessary.
- Destructive operations, such as NCONC, SORT, DELETE,
RPLACA, and RPLACD, should be used carefully and sparingly.
In general, trust the garbage collector: allocate new
data structures when you need them.
To improve the readability of your code,
- Don't use any C{A,D}R functions with more than two
letters between the C and the R. When nested, they become
hard to read. If you have complex data structures, you
are often better off describing them with a DEFSTRUCT,
even if the type is LIST. The data abstraction afforded by
DEFSTRUCT makes the code much more readable and its purpose
clearer. If you must use C{A,D}R, try to use
DESTRUCTURING-BIND instead, or at least SECOND, THIRD,
NTH, NTHCDR, etc.
- Use COND instead of IF and PROGN. In general, don't use PROGN if
there is a way to write the code within an implicit
PROGN. For example,

```
(IF (FOO X)
(PROGN (PRINT "hi there") 23)
34)
```

should be written using COND instead.
- Never use a 2-argument IF or a 3-argument IF with a second
argument of NIL unless you want to emphasize the return value;
use WHEN and UNLESS instead. You will want to emphasize the
return value when the IF clause is embedded within a SETQ,
such as (SETQ X (IF (EQ Y Z) 2 NIL)). If the second argument

to IF is the same as the first, use OR instead: (OR P Q) rather
than (IF P P Q). Use UNLESS instead of (WHEN (NOT ..) ..)
but not instead of (WHEN (NULL ..) ..).
- Use COND instead of nested IF statements. Be sure to check for
unreachable cases, and eliminate those cond-clauses.
- Use backquote, rather than explicit calls to LIST, CONS, and
APPEND, whenever writing a form which produces a Lisp form, but
not as a general substitute for LIST, CONS and APPEND. LIST,
CONS and APPEND usually allocate new storage, but lists produced
by backquote may involve destructive modification (e.g., ,.).
- Make the names of special (global) variables begin and end
with an asterisk (*): (DEFVAR *GLOBAL-VARIABLE*)
Some programmers will mark the beginning and end of an internal
global variable with a percent (%) or a period (.).
Make the names of constants begin and end with a plus (+):
(DEFCONSTANT +E+ 2.7182818)
This helps distinguish them from lexical variables. Some people
prefer to use macros to define constants, since this avoids
the problem of accidentally trying to bind a symbol declared
with defconstant.
- If your program is built upon an underlying substrate which is
implementation-dependent, consider naming those functions and
macros in a way that visually identifies them, either by placing
them in their own package, or prepending a character like a %, .,
or ! to the function name. Note that many programmers use the
$ as a macro character for slot access, so it should be avoided
unless you're using it for that purpose.
- Don't use property lists. Instead, use an explicit hash table.
This helps avoid problems caused by the symbol being in the wrong
package, accidental reuse of property keys from other
programs, and allows you to customize the structure of the table.
- Use the most specific construct that does the job. This lets
readers of the code see what you intended when writing the code.
For example, don't use SETF if SETQ will do (e.g., for lexical
variables). Using SETQ will tell readers of your code that you
aren't doing anything fancy. Likewise, don't use EQUAL where EQ
will do. Use the most specific predicate to test your conditions.
- If you intend for a function to be a predicate, have it return T
for true, not just non-NIL. If there is nothing worth returning
from a function, returning T is conventional. But if a function
is intended to be more than just a predicate, it is better to
return a useful value. (For example, this is one of the differences
between MEMBER and FIND.)
- When NIL is used as an empty list, use () in your code. When NIL
is used as a boolean, use NIL. Similarly, use NULL to test for an
empty list, NOT to test a logical value. Use ENDP to test for the
end of a list, not NULL.
- Don't use the &AUX lambda-list keyword. It is always clearer to
define local variables using LET or LET*.
- When using RETURN and RETURN-FROM to exit from a block, don't
use (VALUES ..) when returning only one value, except if you
are using it to suppress extra multiple values from the first
argument.
- If you want a function to return no values (i.e., equivalent to
VOID in C), use (VALUES) to return zero values. This signals
to the reader that the function is used mainly for side-effects.
- (VALUES (VALUES 1 2 3)) returns only the first value, 1.
You can use (VALUES (some-multiple-value-function ..)) to suppress
the extra multiple values from the function. Use MULTIPLE-VALUE-PROG1
instead of PROG1 when the multiple values are significant.
- When using MULTIPLE-VALUE-BIND and DESTRUCTURING-BIND, don't rely
on the fact that NIL is used when values are missing. This is
an error in some implementations of DESTRUCTURING-BIND. Instead,
make sure that your function always returns the proper number of
values.
- Type the name of external symbols, functions, and variables
from the COMMON-LISP package in uppercase. This will allow your
code to work properly in a case-sensitive version of Common Lisp,
since the print-names of symbols in the COMMON-LISP package
are uppercase internally. (However, not everybody feels that
being nice to case-sensitive Lisps is a requirement, so this
isn't an absolute style rule, just a suggestion.)
Lisp Idioms:
- MAPCAN is used with a function to return a variable number of
items to be included in an output list. When the function returns zero
or one items, the function serves as a filter. For example,
(mapcan #'(lambda (x) (when (and (numberp x) (evenp x)) (list x)))
'(1 2 3 4 x 5 y 6 z 7))
Documentation:
- Comment your code. Use three semicolons in the left margin before
the definition for major explanations. Use two semicolons that
float with the code to explain the routine that follows. Two
semicolons may also be used to explain the following line when the

comment is too long for the single semicolon treatment. Use
a single semicolon to the right of the code to explain a particular
line with a short comment. The number of semicolons used roughly
corresponds with the length of the comment. Put at least one blank
line before and after top-level expressions.
- Include documentation strings in your code. This lets users
get help while running your program without having to resort to
the source code or printed documentation.
Issues related to macros:
- Never use a macro instead of a function for efficiency reasons.
Declaim the function as inline -- for example,
(DECLAIM (INLINE ..))
This is *not* a magic bullet -- be forewarned that inline
expansions can often increase the code size dramatically. INLINE
should be used only for short functions where the tradeoff is
likely to be worthwhile: inner loops, types that the compiler
might do something smart with, and so on.
- When defining a macro that provides an implicit PROGN, use the
&BODY lambda-list keyword instead of &REST.
- Use gensyms for bindings within a macro, unless the macro lets
the user explicitly specify the variable. For example:

```
(defmacro foo ((iter-var list) body-form &body body)
(let ((result (gensym "RESULT")))
`(let ((,result nil))
(dolist (,iter-var ,list ,result)
(setq ,result ,body-form)
(when ,result
,@body)))))
```

This avoids errors caused by collisions during macro expansion
between variable names used in the macro definition and in the
supplied body.
- Use a DO- prefix in the name of a macro that does some kind of
iteration, WITH- when the macro establishes bindings, and
DEFINE- or DEF- when the macro creates some definitions. Don't
use the prefix MAP- in macro names, only in function names.
- Don't create a new iteration macro when an existing function
or macro will do.
- Don't define a macro where a function definition will work just
as well -- remember, you can FUNCALL or MAPCAR a function but
not a macro.
- The LOOP and SERIES macros generate efficient code. If you're
writing a new iteration macro, consider learning to use one
of them instead.
File Modularization:
- If your program involves macros that are used in more than one
file, it is generally a good idea to put such macros in a separate
file that gets loaded before the other files. The same things applies
to primitive functions. If a macro is complicated, the code that
defines the macro should be put into a file by itself. In general, if
a set of definitions form a cohesive and "independent" whole, they
should be put in a file by themselves, and maybe even in their own
package. It isn't unusual for a large Lisp program to have files named
"site-dependent-code", "primitives.lisp", and "macros.lisp". If a file
contains primarily macros, put "-macros" in the name of the file.
Stylistic preferences:
- Use (SETF (CAR ..) ..) and (SETF (CDR ..) ..) in preference to
RPLACA and RPLACD. Likewise (SETF (GET ..) ..) instead of PUT.
- Use INCF, DECF, PUSH and POP instead instead of the corresponding
SETF forms.
- Many programmers religiously avoid using CATCH, THROW, BLOCK,
PROG, GO and TAGBODY. Tags and go-forms should only be necessary
to create extremely unusual and complicated iteration constructs. In
almost every circumstance, a ready-made iteration construct or
recursive implementation is more appropriate.
- Don't use LET* where LET will do. Don't use LABELS where FLET
will do. Don't use DO* where DO will do.
- Don't use DO where DOTIMES or DOLIST will do.
- If you like using MAPCAR instead of DO/DOLIST, use MAPC when
no result is needed -- it's more efficient, since it doesn't
cons up a list. If a single cumulative value is required, use
REDUCE. If you are seeking a particular element, use FIND,
POSITION, or MEMBER.
- If using REMOVE and DELETE to filter a sequence, don't use the
:test-not keyword or the REMOVE-IF-NOT or DELETE-IF-NOT functions.
Use COMPLEMENT to complement the predicate and the REMOVE-IF
or DELETE-IF functions instead.
- Use complex numbers to represent points in a plane.
- Don't use lists where vectors are more appropriate. Accessing the
nth element of a vector is faster than finding the nth element
of a list, since the latter requires pointer chasing while the
former requires simple addition. Vectors also take up less space
than lists. Use adjustable vectors with fill-pointers to
implement a stack, instead of a list -- using a list continually

conses and then throws away the conses.
- When adding an entry to an association list, use ACONS, not
two calls to CONS. This makes it clear that you're using an alist.
- If your association list has more than about 10 entries in it,
consider using a hash table. Hash tables are often more efficient.
(See also [2-2].)
- When you don't need the full power of CLOS, consider using
structures instead. They are often faster, take up less space, and
easier to use.
- Use PRINT-UNREADABLE-OBJECT when writing a print-function.
- Use WITH-OPEN-FILE instead of OPEN and CLOSE.
- When a HANDLER-CASE clause is executed, the stack has already
unwound, so dynamic bindings that existed when the error
occured may no longer exist when the handler is run. Use
HANDLER-BIND if you need this.
- When using CASE and TYPECASE forms, if you intend for the form
to return NIL when all cases fail, include an explicit OTHERWISE
clause. If it would be an error to return NIL when all cases
fail, use ECASE, CCASE, ETYPECASE or CTYPECASE instead.
- Use local variables in preference to global variables whenever
possible. Do not use global variables in lieu of parameter passing.
Global variables can be used in the following circumstances:
* When one function needs to affect the operation of
another, but the second function isn't called by the first.
(For example, *load-pathname* and *break-on-warnings*.)
* When a called function needs to affect the current or future
operation of the caller, but it doesn't make sense to accomplish
this by returning multiple values.
* To provide hooks into the mechanisms of the program.
(For example, *evalhook*, *, /, and +.)
* Parameters which, when their value is changed, represent a
major change to the program.
(For example, *print-level* and *print-readably*.)
* For state that persists between invocations of the program.
Also, for state which is used by more than one major program.
(For example, *package*, *readtable*, *gensym-counter*.)
* To provide convenient information to the user.
(For example, *version* and *features*.)
* To provide customizable defaults.
(For example, *default-pathname-defaults*.)
* When a value affects major portions of a program, and passing
this value around would be extremely awkward. (The example
here is output and input streams for a program. Even when
the program passes the stream around as an argument, if you
want to redirect all output from the program to a different
stream, it is much easier to just rebind the global variable.)
- Beginning students, especially ones accustomed to programming
in C, Pascal, or Fortran, tend to use global variables to hold or pass
information in their programs. This style is considered ugly by
experienced Lisp programmers. Although assignment statements can't
always be avoided in production code, good programmers take advantage
of Lisp's functional programming style before resorting to SETF and
SETQ. For example, they will nest function calls instead of using a
temporary variable and use the stack to pass multiple values. When
first learning to program in Lisp, try to avoid SETF/SETQ and their
cousins as much as possible. And if a temporary variable is necessary,
bind it to its first value in a LET statement, instead of letting it
become a global variable by default. (If you see lots of compiler
warnings about declaring variables to be special, you're probably
making this mistake. If you intend a variable to be global, it should
be defined with a DEFVAR or DEFPARAMETER statement, not left to the
compiler to fix.)
Correctness and efficiency issues:
- In CLtL2, IN-PACKAGE does not evaluate its argument. Use defpackage
to define a package and declare the external (exported)
symbols from the package.
- The ARRAY-TOTAL-SIZE-LIMIT may be as small as 1024, and the
CALL-ARGUMENTS-LIMIT may be as small as 50.
- Novices often mistakenly quote the conditions of a CASE form.
For example, (case x ('a 3) ..) is incorrect. It would return
3 if x were the symbol QUOTE. Use (case x (a 3) ..) instead.
- Avoid using APPLY to flatten lists. Although
(apply #'append list-of-lists)
may look like a call with only two arguments, it becomes a
function call to APPEND, with the LIST-OF-LISTS spread into actual
arguments. As a result it will have as many arguments as there are
elements in LIST-OF-LISTS, and hence may run into problems with the
CALL-ARGUMENTS-LIMIT. Use REDUCE or MAPCAN instead:
(reduce #'append list-of-lists :from-end t)
(mapcan #'copy-list list-of-lists)
The second will often be more efficient (see note below about choosing
the right algorithm). Beware of calls like (apply f (mapcar ..)).
- NTH must cdr down the list to reach the elements you are

interested in. If you don't need the structural flexibility of
lists, try using vectors and the ELT function instead.
- CASE statements can be vectorized if the keys are consecutive
numbers. Such CASE statements can still have OTHERWISE clauses.
To take advantage of this without losing readability, use #. with
symbolic constants:

```
(eval-when (compile load eval)
(defconstant RED 1)
(defconstant GREEN 2)
(defconstant BLUE 3))
(case color
(#.RED ...)
(#.GREEN ...)
(#.BLUE ...)
...)
```

- Don't use quoted constants where you might later destructively
modify them. For example, instead of writing '(c d) in

```
(defun foo ()
(let ((var '(c d)))
..))
```

write (list 'c 'd) instead. Using a quote here can lead to
unexpected results later. If you later destructively modify the
value of var, this is self-modifying code! Some Lisp compilers
will complain about this, since they like to make constants
read-only. Modifying constants has undefined results in ANSI CL.
See also the answer to question [3-13].
Similarly, beware of shared list structure arising from the use
of backquote. Any sublist in a backquoted expression that doesn't
contain any commas can share with the original source structure.
- Don't proclaim unsafe optimizations, such as

```
(proclaim '(optimize (safety 0) (speed 3) (space 1)))
```

since this yields a global effect. Instead, add the
optimizations as local declarations to small pieces of
well-tested, performance-critical code:

```
(defun well-tested-function ()
(declare (optimize (safety 0) (speed 3) (space 1)))
..)
```

Such optimizations can remove run-time type-checking; type-checking
is necessary unless you've very carefully checked your code
and added all the appropriate type declarations.
- Some programmers feel that you shouldn't add declarations to
code until it is fully debugged, because incorrect
declarations can be an annoying source of errors. They recommend
using CHECK-TYPE liberally instead while you are developing the code.
On the other hand, if you add declarations to tell the
compiler what you think your code is doing, the compiler can
then tell you when your assumptions are incorrect.
Declarations also make it easier for another programmer to read
your code.
- Declaring the type of variables to be FIXNUM does not
necessarily mean that the results of arithmetic involving the
fixnums will be a fixnum; it could be a BIGNUM. For example,

```
(declare (type fixnum x y))
(setq z (+ (* x x) (* y y)))
```

could result in z being a BIGNUM. If you know the limits of your
numbers, use a declaration like

```
(declare (type (integer 0 100) x y))
```

instead, since most compilers can then do the appropriate type
inference, leading to much faster code.
- Don't change the compiler optimization with an OPTIMIZE
proclamation or declaration until the code is fully debugged
and profiled. When first writing code you should say
(declare (optimize (safety 3))) regardless of the speed setting.
- Depending on the optimization level of the compiler, type
declarations are interpreted either as (1) a guarantee from
you that the variable is always bound to values of that type,
or (2) a desire that the compiler check that the variable is
always bound to values of that type. Use CHECK-TYPE if (2) is
your intention.
- If you get warnings about unused variables, add IGNORE
declarations if appropriate or fix the problem. Letting such
warnings stand is a sloppy coding practice.
To produce efficient code,
- choose the right algorithm. For example, consider seven possible
implementations of COPY-LIST:

```
(defun copy-list (list)
(let ((result nil))
(dolist (item list result)
(setf result (append result (list item))))))
(defun copy-list (list)
(let ((result nil))
(dolist (item list (nreverse result))
(push item result))))
```

```
(defun copy-list (list)
(mapcar #'identity list))
(defun copy-list (list)
(let ((result (make-list (length list))))
(do ((original list (cdr original))
(new result (cdr new)))
((null original) result)
(setf (car new) (car original)))))
(defun copy-list (list)
(when list
(let* ((result (list (car list)))
(tail-ptr result))
(dolist (item (cdr list) result)
(setf (cdr tail-ptr) (list item))
(setf tail-ptr (cdr tail-ptr))))))
(defun copy-list (list)
(loop for item in list collect item))
(defun copy-list (list)
(if (consp list)
(cons (car list)
(copy-list (cdr list)))
list))
```

The first uses APPEND to tack the elements onto the end of the list. Since APPEND must traverse the entire partial list at each step, this yields a quadratic running time for the algorithm. The second implementation improves on this by iterating down the list twice; once to build up the list in reverse order, and the second time to reverse it. The efficiency of the third depends on the Lisp implementation, but it is usually similar to the second, as is the fourth. The fifth algorithm, however, iterates down the list only once. It avoids the extra work by keeping a pointer (reference) to the last cons of the list and RPLACDing onto the end of that. Use of the fifth algorithm may yield a speedup. Note that this contradicts the earlier dictum to avoid destructive functions. To make more efficient code one might selectively introduce destructive operations in critical sections of code. Nevertheless, the fifth implementation may be less efficient in Lisps with cdr-coding, since it is more expensive to RPLACD cdr-coded lists. Depending on the implementation of nreverse, however, the fifth and second implementations may be doing the same amount of work. The sixth example uses the Loop macro, which usually expands into code similar to the third. The seventh example copies dotted lists, and runs in linear time, but isn't tail-recursive.
There is a long-running discussion of whether pushing items onto a list and then applying NREVERSE to the result is faster or slower than the alternatives. According to Richard C. Waters (Lisp Pointers VI(4):27-34, October-December 1993), the NREVERSE strategy is slightly faster in most Lisp implementations. But the speed difference either way isn't much, so he argues that one should pursue the option that yields the clearest and simplest code, namely using NREVERSE.
Here's code for a possible implementation of NREVERSE. As is evident, most of the alternatives to using NREVERSE involve essentially the same code, just reorganized.

```
(defun nreverse (list)
;; REVERSED is the partially reversed list,
;; CURRENT is the current cons cell, which will be reused, and
;; REMAINING are the cons cells which have not yet been reversed.
(do* ((reversed nil)
(current list remaining)
(remaining (cdr current) (cdr current)))
((null current)
reversed)
;; Reuse the cons cell at the head of the list:
;; reversed := ((car remaining) . reversed)
(setf (cdr current) reversed)
(setf reversed current)))
```

- use type declarations liberally in time-critical code, but only if you are a seasoned Lisp programmer. Appropriate type declarations help the compiler generate more specific and optimized code. It also lets the reader know what assumptions were made. For example, if you only use fixnum arithmetic, adding declarations can lead to a significant speedup. If you are a novice Lisp programmer, you should use type declarations sparingly, as there may be no checking to see if the declarations are correct, and optimized code can be harder to debug. Wrong declarations can lead to errors in otherwise correct code, and can limit the reuse of code in other contexts. Depending on the Lisp compiler, it may also be necessary to declare the type of results using THE, since some compilers don't deduce the result type from the inputs.

Read More Answers.

Question # 7

How we have two Lisp processes communicate via unix sockets?

Answer:-

CLX uses Unix sockets to communicate with the X window server. Look at the following files from the CLX distribution for a good example of using Unix sockets from Lisp:
defsystem.lisp Lucid, AKCL, IBCL, CMU.
socket.c, sockcl.lisp AKCL, IBCL
excldep.lisp Franz Allegro CL
You will need the "socket.o" files which come with Lucid and Allegro.
To obtain CLX

Read More Answers.

Question # 8

What is the difference between Scheme and Common Lisp?

Answer:-

Scheme is a dialect of Lisp that stresses conceptual elegance and simplicity. It is specified in R4RS and IEEE standard P1178. (See the Scheme FAQ for details on standards for Scheme.) Scheme is much smaller than Common Lisp; the specification is about 50 pages, compared to Common Lisp's 1300 page draft standard. (See question [4-10] for details on standards for Common Lisp.) Advocates of Scheme often find it amusing that the Scheme standard is shorter than the index to CLtL2.
Scheme is often used in computer science curricula and programming language research, due to its ability to represent many programming abstractions with its simple primitives. Common Lisp is often used for real world programming because of its large library of utility functions, a standard object-oriented programming facility (CLOS), and a sophisticated condition handling system.
See the Scheme FAQ for information about object-oriented programming in Scheme.
In Common Lisp, a simple program would look something like the following:

```
(defun fact (n)
(if (< n 2)
1
(* n (fact (1- n)))))
```

In Scheme, the equivalent program would like like this:

```
(define fact
(lambda (n)
(if (< n 2)
1
(* n (fact (- n 1))))))
```

Experienced Lisp programmers might write this program as follows in order to allow it to run in constant space:

```
(defun fact (n)
(labels ((tail-recursive-fact (counter accumulator)
(if (> counter n)
accumulator
(tail-recursive-fact (1+ counter)
(* counter accumulator)))))
(tail-recursive-fact 1 1)))
```

Whereas in Scheme the same computation could be written as follows:

```
(define fact
(lambda (n)
(letrec ((tail-recursive-fact
(lambda (counter accumulator)
(if (> counter n)
accumulator
(tail-recursive-fact (+ counter 1)
(* counter accumulator))))))
(tail-recursive-fact 1 1))))
```

or perhaps (using IEEE named LETs):

```
(define fact
(lambda (n)
(let loop ((counter n)
(accumulator 1))
(if (< counter 2)
accumulator
(loop (- counter 1)
(* accumulator counter))))))
```

Some Schemes allow one to use the syntax (define (fact n) ...) instead of (define fact (lambda (n) ...)).

Read More Answers.

Question # 9

Can we call Lisp functions from other languages?

Answer:-

In implementations that provide a foreign function interface as described
above, there is also usually a "callback" mechanism. The programmer may
associate a foreign language function name with a Lisp function. When a
foreign object file or library is loaded into the Lisp address space, it is
linked with these callback functions. As with foreign functions, the
programmer must supply the argument and result data types so that Lisp may
perform conversions at the interface. Note that in such foreign function
interfaces Lisp is often left "in control" of things like memory
allocation, I/O channels, and startup code (this is a major nuisance
for lots of people).

Read More Answers.

Question # 10

How to determine if a file is a directory or not? How do I get the current directory name from within a Lisp program? Is there any way to create a directory?

Answer:-

There is no portable way in Common Lisp of determining whether a file
is a directory or not. Calling DIRECTORY on the pathname will not
always work, since the directory could be empty. For UNIX systems
(defun DIRECTORY-P (pathname)
(probe-file (concatenate 'string pathname "/.")))
seems to work fairly reliably. (If "foo" is a directory, then "foo/."
will be a valid filename; if not, it will return NIL.) This won't, of
course, work on the Macintosh, or on other operating systems (e.g.,
MVS, CMS, ITS). On the Macintosh, use DIRECTORYP.
Moreover, some operating systems may not support the concept of
directories, or even of a file system. For example, recent work on
object-oriented technology considers files to be collections of
objects. Each type of collection defines a set of methods for reading
and writing the objects "stored" in the collection.
There's no standard function for finding the current directory from
within a Lisp program, since not all Lisp environments have the
concept of a current directory. Here are the commands from some Lisp
implementations:
Lucid: WORKING-DIRECTORY (which is also SETFable)
PWD and CD also work
Allegro: CURRENT-DIRECTORY (use excl:chdir to change it)
CMU CL: DEFAULT-DIRECTORY
LispWorks: LW:*CURRENT-WORKING-DIRECTORY*
(use LW:CHANGE-DIRECTORY to change it)
Allegro also uses the variable *default-pathname-defaults* to resolve
relative pathnames, maintaining it as the current working directory.
So evaluating (truename "./") in Allegro (and on certain other
systems) will return a pathname for the current directory. Likewise,
in some VMS systems evaluating (truename "[]") will return a pathname
for the current directory.

Read More Answers.

Question # 11

Why we need LISP?

Answer:-

In short, no single improvement you can make to your AutoCAD system will save you more time, effort, and money. You can spend thousands on the latest generation computers, the fastest video cards, and so on, but that won't make nearly as big a difference as automating your system with software. With an arsenal of LISP routines, you will send accuracy, consistency, and productivity soaring while greatly reducing the stress and strain of CAD operation. The right software is the key, and there is no amount of software you can buy that beats being able to program AutoCAD yourself with as many tailor-made routines as you want. And that is exactly what the LISP Generator enables you to do.

Read More Answers.

Question # 12

How to save an executable image of my loaded Lisp system? How to run a Unix command in Lisp? How to exit Lisp? Access environment variables?

Answer:-

There is no standard for dumping a Lisp image. Here are the
commands from some lisp implementations:
Lucid: DISKSAVE
Symbolics: Save World [CP command]
CMU CL: SAVE-LISP
Franz Allegro: EXCL:DUMPLISP (documented)
SAVE-IMAGE (undocumented)
Medley: IL:SYSOUT or IL:MAKESYS
MCL: SAVE-APPLICATION <pathname>
&key :toplevel-function :creator :excise-compiler
:size :resources :init-file :clear-clos-caches
KCL: (si:save-system "saved_kcl")
LispWorks: LW:SAVE-IMAGE
Be sure to garbage collect before dumping the image. You may need to

experiment with the kind of garbage collection for large images, and
may find better results if you build the image in stages.
There is no standard for running a Unix shell command from Lisp,
especially since not all Lisps run on top of Unix. Here are the
commands from some Lisp implementations:
Allegro: EXCL:RUN-SHELL-COMMAND (command &key input output
error-output wait if-input-does-not-exist
if-output-exists if-error-output-exists)
Lucid: RUN-PROGRAM (name
&key input output
error-output (wait t) arguments
(if-input-does-not-exist :error)
(if-output-exists :error)
(if-error-output-exists :error))
KCL: SYSTEM
For example, (system "ls -l").
You can also try RUN-PROCESS and EXCLP, but they
don't work with all versions of KCL.
CMU CL: RUN-PROGRAM (program args
&key (env *environment-list*) (wait t) pty input
if-input-does-not-exist output
(if-output-exists :error) (error :output)
(if-error-exists :error) status-hook before-execve)
LispWorks: FOREIGN:CALL-SYSTEM-SHOWING-OUTPUT
To toggle source file recording and cross-reference annotations, use
Allegro: excl:*record-source-file-info*
excl:*load-source-file-info*
excl:*record-xref-info*
excl:*load-xref-info*
LispWorks: (toggle-source-debugging nil)
Memory management:
CMU CL: (bytes-consed-between-gcs) [this is setfable]
Lucid: (change-memory-management
&key growth-limit expand expand-reserved)
Allegro: *tenured-bytes-limit*
LispWorks: LW:GET-GC-PARAMETERS
(use LW:SET-GC-PARAMETERS to change them)
Environment Variable Access:
Allegro: (sys:getenv var)
(sys:setenv var value) or (setf (sys:getenv var) value)
Lucid: (environment-variable var)
(set-environment-variable var value)
CMU CL 17: (cdr (assoc (intern var :keyword) *environment-list*))
{A}KCL, GCL: (system:getenv var)
CLISP: (system::getenv var)
Exiting/Quitting:
CLISP: EXIT
Allegro: EXIT (&optional excl::code &rest excl::args
&key excl::no-unwind excl::quiet)
LispWorks: BYE (&optional (arg 0))
Lucid: QUIT (&optional (lucid::status 0))
CMU CL: QUIT (&optional recklessly-p)

Read More Answers.

Question # 13

How to save programs to files?

Answer:-

files are standard LISP code that will run on any AutoCAD system.

Read More Answers.

Question # 14

How to call non-Lisp functions from Lisp?

Answer:-

Most Lisp implementations for systems where Lisp is not the most common
language provide a "foreign function" interface. As of now there has been
no significant standardization effort in this area. They tend to be
similar, but there are enough differences that it would be inappropriate to
try to describe them all here. In general, one uses an
implementation-dependent macro that defines a Lisp function, but instead of
supplying a body for the function, one supplies the name of a function written
in another language; the argument list portion of the definition is
generally augmented with the data types the foreign function expects and
the data type of the foreign function's return value, and the Lisp
interface function arranges to do any necessary conversions. There is also
generally a function to "load" an object file or library compiled in a
foreign language, which dynamically links the functions in the file being
loaded into the address space of the Lisp process, and connects the
interface functions to the corresponding foreign functions.

If you need to do this, see the manual for your language implementation for
full details. In particular, be on the lookout for restrictions on the
data types that may be passed. You may also need to know details about the
linkage conventions that are used on your system; for instance, many C
implementations prepend an underscore onto the names of C functions when
generating the assembler output (this allows them to use names without
initial underscores internally as labels without worrying about conflicts),
and the foreign function interface may require you to specify this form
explicitly.
Franz Allegro Common Lisp's "Foreign Function Call Facility" is
described in chapter 10 of the documentation. Calling Lisp Functions
from C is treated in section 10.8.2. The foreign function interface in
Macintosh Common Lisp is similar. The foreign function interface for
KCL is described in chapter 10 of the KCL Report. The foreign function
interfaces for Lucid on the Vax and Lucid on the Sun4 are
incompatible. Lucid's interface is described in chapter 5 of the
Advanced User's Guide.

Read More Answers.

Question # 15

What is a "Lisp Machine" (LISPM)?

Answer:-

A Lisp machine (or LISPM) is a computer which has been optimized to run lisp
efficiently and provide a good environment for programming in it. The
original Lisp machines were implemented at MIT, with spinoffs as LMI (defunct)
and Symbolics (bankrupt). Xerox also had a series of Lisp machines
(Dandylion, Dandytiger), as did Texas Instruments (TI Explorer). The
TI and Symbolics Lisp machines are currently available as cards that
fit into Macintosh computers (the so-called "Lisp on a chip").
Optimizations typical of Lisp machines include:
- Hardware Type Checking. Special type bits let the type be checked
efficiently at run-time.
- Hardware Garbage Collection.
- Fast Function Calls.
- Efficient Representation of Lists.
- System Software and Integrated Programming Environments.
For further information, see:
Paul Graham, "Anatomy of a Lisp Machine", AI Expert, December 1988.
Pleszkun and Thazhuthaveetil, "The Architecture of Lisp Machines",
IEEE Computer, March 1987.
Ditzel, Schuler and Thomas, "A Lisp Machine Profile: Symbolics 3650",
AI Expert, January 1987.
Peter M. Kogge, "The Architecture of Symbolic Computers",
McGraw-Hill 1991. ISBN 0-07-035596-7.

Read More Answers.

Question # 16

Explain What is the ouput like?

Answer:-

LG3 writes easy-to-read, fully indented AutoLISP programs loaded with detailed comments in English that explain what all of the LISP code is doing. The files are standard ".LSP" files. You can view, edit, or print them with any editor, and run them on any AutoCAD system with or without the Generator.

Read More Answers.

Question # 17

Who is the founder of Lisp Programming?

Answer:-

John McCarthy developed the basics behind Lisp during the 1956 Dartmouth
Summer Research Project on Artificial Intelligence. He intended it as an
algebraic LISt Processing (hence the name) language for artificial
intelligence work. Early implementations included the IBM 704, the IBM
7090, the DEC PDP-1, the DEC PDP-6 and the DEC PDP-10. The PDP-6 and
PDP-10 had 18-bit addresses and 36-bit words, allowing a CONS cell to
be stored in one word, with single instructions to extract the CAR and
CDR parts. The early PDP machines had a small address space, which
limited the size of Lisp programs.
Milestones in the development of Lisp:
1956 Dartmouth Summer Research Project on AI.
1960-65 Lisp1.5 is the primary dialect of Lisp.
1964- Development of BBNLisp at BBN.
late 60s Lisp1.5 diverges into two main dialects:
Interlisp (originally BBNLisp) and MacLisp.
early 70s Development of special-purpose computers known as Lisp
Machines, designed specificly to run Lisp programs.
Xerox D-series Lisp Machines run Interlisp-D.
Early MIT Lisp Machines run Lisp Machine Lisp
(an extension of MacLisp).

1969 Anthony Hearn and Martin Griss define Standard Lisp to
port REDUCE, a symbolic algebra system, to a variety
of architectures.
late 70s Macsyma group at MIT developed NIL (New Implementation
of Lisp), a Lisp for the VAX.
Stanford and Lawrence Livermore National Laboratory
develop S-1 Lisp for the Mark IIA supercomputer.
Franz Lisp (dialect of MacLisp) runs on stock-hardware
Unix machines.
Gerald J. Sussman and Guy L. Steele developed Scheme,
a simple dialect of Lisp with lexical scoping and
lexical closures, continuations as first-class objects,
and a simplified syntax (i.e., only one binding per symbol).
Advent of object-oriented programming concepts in Lisp.
Flavors was developed at MIT for the Lisp machine,
and LOOPS (Lisp Object Oriented Programming System) was
developed at Xerox.
early 80s Development of SPICE-Lisp at CMU, a dialect of MacLisp
designed to run on the Scientific Personal Integrated
Computing Environment (SPICE) workstation.
1980 First biannual ACM Lisp and Functional Programming Conf.
1981 PSL (Portable Standard Lisp) runs on a variety of platforms.
1981+ Lisp Machines from Xerox, LMI (Lisp Machines Inc)
and Symbolics available commercially.
April 1981 Grass roots definition of Common Lisp as a description
of the common aspects of the family of languages (Lisp
Machine Lisp, MacLisp, NIL, S-1 Lisp, Spice Lisp, Scheme).
1984 Publication of CLtL1. Common Lisp becomes a de facto
standard.
1986 X3J13 forms to produce a draft for an ANSI Common Lisp
standard.
1987 Lisp Pointers commences publication.
1990 Steele publishes CLtL2 which offers a snapshot of
work in progress by X3J13. (Unlike CLtL1, CLtL2
was NOT an output of the standards process and was
not intended to become a de facto standard. Read
the Second Edition Preface for further explanation
of this important issue.) Includes CLOS,
conditions, pretty printing and iteration facilities.
1992 X3J13 creates a draft proposed American National
Standard for Common Lisp. This document is the
first official successor to CLtL1.
[Note: This summary is based primarily upon the History section of the
draft ANSI specification.

Read More Answers.

Question # 18

Explain the purpose of this newsgroup?

Answer:-

The newsgroup comp.lang.lisp exists for general discussion of
topics related to the programming language Lisp. For example, possible
topics can include (but are not necessarily limited to):
announcements of Lisp books and products
discussion of programs and utilities written in Lisp
discussion of portability issues
questions about possible bugs in Lisp implementations
problems porting an implementation to some architecture
Postings should be of general interest to the Lisp community. See also
question [4-9]. Postings asking for solutions to homework problems are
inappropriate.
The comp.lang.lisp newsgroup is archived in
ftp.cs.cmu.edu:/user/ai/pubs/news/comp.lang.lisp/
on a weekly basis.
Every so often, somebody posts an inflammatory message, such as
My programming language is better than yours (Lisp vs. C/Prolog/Scheme).
Loop (or Series) should/shouldn't be part of the language.
These "religious" issues serve no real purpose other than to waste
bandwidth. If you feel the urge to respond to such a post, please do
so through a private e-mail message.
Questions about object oriented programming in Lisp should be directed
to the newsgroup comp.lang.clos. Similarly, questions about the
programming language Scheme should be directed to the newsgroup
comp.lang.scheme. Discussion of functional programming language issues
should be directed to the newsgroup comp.lang.functional. Discussion
of AI programs implemented in Lisp should sometimes be cross-posted to
the newsgroup comp.ai.

Read More Answers.

Question # 19

How to learn about implementing Lisp interpreters and compilers?

Answer:-

Books about Lisp implementation include:
1. John Allen
"Anatomy of Lisp"
McGraw-Hill, 1978. 446 pages. ISBN 0-07-001115-X
Discusses some of the fundamental issues involved in
the implemention of Lisp.
2. Samuel Kamin
"Programming Languages, An Interpreter-Based Approach"
Addison-Wesley, Reading, Mass., 1990. ISBN 0-201-06824-9
Includes sources to several interpreters for Lisp-like languages.
The source for the interpreters in the book is available
by anonymous FTP from
a.cs.uiuc.edu:/pub/kamin/kamin.distr/
Tim Budd reimplemented the interpreters in C++, and has made
them available by anonymous ftp from
cs.orst.edu:/pub/budd/kamin/
3. Sharam Hekmatpour
"Lisp: A Portable Implementation"
Prentice Hall, 1985. ISBN 0-13-537490-X.
Describes a portable implementation of a small dynamic
Lisp interpreter (including C source code).
4. Peter Henderson
"Functional Programming: Application and Implementation"
Prentice-Hall (Englewood Cliffs, NJ), 1980. 355 pages.
5. Peter M. Kogge
"The Architecture of Symbolic Computers"
McGraw-Hill, 1991. ISBN 0-07-035596-7.
Includes sections on memory management, the SECD and
Warren Abstract Machines, and overviews of the various
Lisp Machine architectures.
6. Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes
"Essentials of Programming Languages"
MIT Press, 1992, 536 pages. ISBN 0-262-06145-7.
Teaches fundamental concepts of programming language
design by using small interpreters as examples. Covers
most of the features of Scheme. Includes a discussion
of parameter passing techniques, object oriented languages,
and techniques for transforming interpreters to allow
their implementation in terms of any low-level language.
Also discusses scanners, parsers, and the derivation of
a compiler and virtual machine from an interpreter.
Includes a few chapters on converting code into a
continuation passing style.
Source files available by anonymous ftp from
cs.indiana.edu:/pub/eopl/ [129.79.254.191].
7. Peter Lee, editor, "Topics in Advanced Language Implementation",
The MIT Press, Cambridge, Mass., 1991.
Articles relevant to the implementation of functional
programming languages.
8. Also see the proceedings of the biannual ACM Lisp and Functional
Programming conferences, the implementation notes for CMU Common Lisp,
Norvig's book, and SICP (Abelson & Sussman).
9. Christian Queinnec
"Les Langages Lisp"
InterEditions (in French), 1994. 500 pages.
ISBN 2-7296-0549-5, 61-2448-1. (?)
Cambridge University Press (in English), 1996.
ISBN 0-521-56247-3.
The book covers Lisp, Scheme and other related dialects,
their interpretation, semantics and compilation.

Read More Answers.

Computer Programming Most Popular Interview Topics.

1 : PHP Frequently Asked Interview Questions and Answers Guide.

2 : C++ Programming Frequently Asked Interview Questions and Answers Guide.

3 : C Programming Frequently Asked Interview Questions and Answers Guide.

4 : Software engineering Frequently Asked Interview Questions and Answers Guide.

5 : Cobol Frequently Asked Interview Questions and Answers Guide.

6 : Visual Basic (VB) Frequently Asked Interview Questions and Answers Guide.

7 : Socket Programming Frequently Asked Interview Questions and Answers Guide.

8 : Perl Programming Frequently Asked Interview Questions and Answers Guide.

9 : VBA Frequently Asked Interview Questions and Answers Guide.

10 : OOP Frequently Asked Interview Questions and Answers Guide.