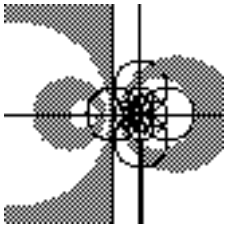


Язык Common Lisp Второе издание

Guy L. Steele Jr.
Thinking Machines Corporation
с участием
Scott E. Fahlman
Carnegie-Mellon University
Richard P. Gabriel
Lucid, Inc.
Stanford University
David A. Moon
Symbolics, Incorporated
Daniel L. Weinreb
Symbolics, Incorporated
и с участием во втором издании
Kent M. Pitman
Symbolics, Incorporated
Richard C. Waters
Massachusetts Institute of Technology
Jon L White
Lucid, Inc.

Опубликовано Digital Press.

Стоит ли удивляться, если под давлением всех упомянутых трудностей конвент был вынужден отойти от искусственной структуры и полной симметрии, которые многомудрый теоретик в угоду своим отвлеченным взглядам предпослал составленной им в кабинете или в воображении конституции?

—*Джеймс Мэдисон, Журнал «Федералист»*
№ 37, 11 января, 1788
http://grachev62.narod.ru/Fed/Fed_37.htm

Содержание

Пролог ВТОРОЕ ИЗДАНИЕ

Common Lisp к успеху пришел. С момента публикации первой редакции данной книги в 1984, много организаций использовали его как *де-факто* стандарт для реализации Lisp'а. В результате сейчас гораздо проще портировать большую Lisp программу с одной реализации на другую. Common Lisp доказал свою полезность и стабильность, как платформы для быстрого прототипирования и быстрой поставки систем в области искусственного интеллекта и не только в ней. С приобретённым опытом использования Common Lisp'а для такого большого количества приложений, организации не нашли недостатков в возможностях для инноваций. Одна из важных характеристик Lisp'а это его хорошая поддержка для экспериментальных расширений языка; несмотря на то, что Common Lisp стабилен, он не инертен.

Версия Common Lisp'а от 1984 года была несовершенной и незавершённой. В некоторых случаях допускались неосторожности: некоторые двузначные ситуации игнорировались и из следствия не определялись, или разные вещи конфликтовали или некоторые свойства Lisp'а были так хорошо известны, что на них традиционно полагались, даже автор забыл их записать. В других случаях неофициальный комитет, что создавал Common Lisp, не мог принять решение и соглашался оставить некоторые важные вещи языка неопределёнными, чем выбирать менее удачный вариант. Например, обработка ошибок; в Common Lisp 1984 года было изобилие способов генерации сигналов об ошибках, но не было методов для их ловушек.

Благодарности ВТОРОЕ ИЗДАНИЕ

Прежде всего, автор должен поблагодарить большое количество людей в Lisp сообществе, которые затратили большой труд для разработки, внедрения и использования Common Lisp. Некоторые из них затратили многие часы усилий в качестве членов комитета ANSI X3J13. Другие сделали презентацию или предложения X3J13, а третьи направили предложения и поправки к первому изданию прямо ко мне. Эта книга основывается на их и моих усилиях.

An early draft of this book was made available to all members of X3J13 for their criticism. I have also worked with the many public documents that have been written during the course of the committee's work (which is not over yet). It is my hope that this book is an accurate reflection of the committee's actions as of October 1989. Nevertheless, any errors or inconsistencies are my responsibility. The fact that I have made a draft available to certain persons, received feedback from them, or thanked them in these acknowledgments does not necessarily imply that any one of them or any of the institutions with which they are affiliated endorse this book or anything of its contents.

Ранний черновик этой книги быть доступен для критики всем членам комитета X3J13. Также автор проработал много опубликованных документов, которые были написаны в течение работы комитета (который ещё не закончился). Автор надеется книги являются наиболее точным отражением работы комитета на октябрь 1989 года. Тем не менее, все ошибки или неточности лежат на ответственности автора.

Digital Press and I gave permission to X3J13 to use any or all parts of the first edition in the production of an ANSI Common Lisp standard. Conversely, in writing this book I have worked with publicly available documents produced by X3J13 in the course of its work, and in some cases as a courtesy

have obtained the consent of the authors of those documents to quote them extensively. This common ancestry will result in similarities between this book and the emerging ANSI Common Lisp standard (that is the purpose, after all). Nevertheless, this second edition has no official connection whatsoever with X3J13 or ANSI, nor is it endorsed by either of those institutions.

Следующие лица были членами X3J13 или участвовали в его деятельности в тот или иной момент: Jim Allard, Dave Andre, Jim Antonisse, William Arbaugh, John Aspinall, Bob Balzer, Gerald Barber, Richard Barber, Kim Barrett, David Bartley, Roger Bate, Alan Bawden, Michael Beckerle, Paul Beiser, Eric Benson, Daniel Bobrow, Mary Boelk, Skona Brittain, Gary Brown, Tom Bucken, Robert Buckley, Gary Byers, Dan Carnese, Bob Cassels, Jérôme Chailloux, Kathy Chapman, Thomas Christaller, Will Clinger, Peter Coffee, John Cugini, Pavel Curtis, Doug Cutting, Christopher Dabrowski, Jeff Dalton, Linda DeMichiel, Fred Discenzo, Jerry Duggan, Patrick Dussud, Susan Ennis, Scott Fahlman, Jogn Fitch, John Foderaro, Richard Gabriel, Steven Gadol, Nick Gall, Oscar Garcia, Robert Giansiracusa, Brad Goldstein, David Gray, Richard Greenblatt, George Hadden, Steve Haflich, Dave Henderson, Carl Hewitt, Carl Hoffman, Cheng Hu, Masayuki Ida, Takayasu Ito, Sonya Keene, James Kempf, Gregory Jennings, Robert Kerns, Gregor Kiczales, Kerry Kimbrough, Dieter Kolb, Timothy Koschmann, Ed Krall, Fritz Kunze, Aaron Larson, Joachim Laubsch, Kevin Layer, Michael Levin, Ray Lim, Thom Linden, David Loeffler, Sandra Loosemore, Barry Margolin, Larry Masinter, David Matthews, Robert Mathis, John McCarthy, Chris McConnell, Rob McLachlan, Jay Mendelsohn, Martin Mikelsons, Tracey Miles, Richard Mlyarnik, David Moon, Jarl Nilsson, Leo Noordhulsen, Ronald Ohlander, Julian Padget, Jeff Peck, Jan Pedersen, Bob Pellegrino, Crispin Perdue, Dan Pierson, Kent Pitman, Dexter Pratt, Christian Quiennec, B. Raghavan, Douglas Rand, Jonathan Rees, Chris Richardson, Jeff Rininger, Walter van Roggen, Jeffrey Rosenking, Don Sakahara, William Scherlis, David Slater, James Smith, Alan Snyder, Angela Sodan, Richard Soley, S. Sridhar, Bill St. Clair, Philip Stanhope, Guy Steele, Herbert Stoyan, Hiroshi Torii, Dave Touretzky, Paul Tucker, Rick Tucker, Thomas Turba, David Unietis, Mary Van Deusen, Ellen Waldrum, Richard Waters, Allen Wechsler, Mark Wegman, Jon L White, Skef Wholey, Alexis Wieland, Martin Yonke, Bill York, Taiichi Yuasa, Gail Zacharias, and Jan Zubkoff.

Автор должен выразить особую благодарность и признательность ряду людей за их выдающиеся усилия:

Larry Masinter, chairman of the X3J13 Cleanup Subcommittee, developed the standard format for documenting all proposals to be voted upon. The result has been an outstanding technical and historical record of all the actions taken by X3J13 to rectify and improve Common Lisp.

Sandra Loosemore, chairwoman of the X3J13 Compiler Subcommittee, produced many proposals for clarifying the semantics of the compilation process. She has been a diligent stickler for detail and has helped to clarify many parts of Common Lisp left vague in the first edition.

Jon L White, chairman of the X3J13 Iteration Subcommittee, supervised the consideration of several controversial proposals, one of which (`loop`) was eventually adopted by X3J13.

Thom Linden, chairman of the X3J13 Character Subcommittee, led a team in grappling with the difficult problem of accommodating various character sets in Common Lisp. One result is that Common Lisp will be more attractive for international use.

Kent Pitman, chairman of the X3J13 Error Handling Subcommittee, plugged the biggest outstanding hole in Common Lisp as described by the first edition.

Kathy Chapman, chairwoman of the X3J13 Drafting Subcommittee, and principal author of the draft standard, has not only written a great deal of text but also insisted on coherent and consistent terminology and pushed the rest of the committee forward when necessary.

Robert Mathis, chairman of X3J13, has kept administrative matters flowing smoothly during technical controversies.

Mary Van Deusen, secretary of X3J13, kept excellent minutes that were a tremendous aid to me in tracing the history of a number of complex discussions.

Jan Zubkoff, X3J13 meeting and mailing organizer, knows what's going on, as always. She is a master of organization and of physical arrangements. Moreover, she once again pulled me out of the fire at the last minute.

Dick Gabriel, international representative for X3J13, has kept information flowing smoothly between Europe, Japan, and the United States. He provided a great deal of the energy and drive for the completion of the Common Lisp Object System specification. He has also provided me with a great deal of valuable advice and has been on call for last-minute consultation at all hours during the final stages of preparation for this book.

David Moon has consistently been a source of reason, expert knowledge, and careful scrutiny. He has read the first edition and the X3J13 proposals

perhaps more carefully than anyone else.

David Moon, Jon L White, Gregor Kiczales, Robert Mathis, Mary Boelk provided extensive feedback on an early draft of this book. I thank them as well as the many others who commented in one way or another on the draft.

I wish to thank the authors of large proposals to X3J13 that have made material available for more or less wholesale inclusion in this book as distinct chapters. This material was produced primarily for the use of X3J13 in its work. It has been included here on a non-exclusive basis with the consent of the authors.

The author of the chapter on `loop` (Jon L White) notes that the chapter is based on documentation written at Lucid, Inc., by Molly M. Miller, Sonia Orin Lyris, and Kris Dinkel. Glenn Burke, Scott Fahlman, Colin Meldrum, David Moon, Cris Perdue, and Dick Waters contributed to the design of the `loop` macro.

The authors of the Common Lisp Object System specification (Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon) wish to thank Patrick Dussud, Kenneth Kahn, Jim Kempf, Larry Masinter, Mark Stefik, Daniel L. Weinreb, and Jon L White for their contributions.

The author of the chapter on Conditions (Kent M. Pitman) notes that there is a paper [38] containing background information about the design of the condition system, which is based on the condition system of the Symbolics Lisp Machines [49]. The members of the X3J13 Error Handling Subcommittee were Andy Daniels and Kent Pitman. Richard Mlynarik and David A. Moon made major design contributions. Useful comments, questions, suggestions, and criticisms were provided by Paul Anagnostopoulos, Alan Bawden, William Chiles, Pavel Curtis, Mary Fontana, Dick Gabriel, Dick King, Susan Lander, David D. Loeffler, Ken Olum, David C. Plummer, Alan Snyder, Eric Weaver, and Daniel L. Weinreb. The Condition System was designed specifically to accommodate the needs of Common Lisp. The design is, however, most directly based on the "New Error System" (NES) developed at Symbolics by David L. Andre, Bernard S. Greenberg, Mike McMahon, David A. Moon, and Daniel L. Weinreb. The NES was in turn based on experiences with the original Lisp Machine error system (developed at MIT), which was found to be inadequate for the needs of the modern Lisp Machine environments. Many aspects of the NES were inspired by the (PL/I) condition system used by the Honeywell Multics operating system. Henry Lieberman provided conceptual guidance and encouragement in the design of the NES. A

reimplementation of the NES for non-Symbolics Lisp Machine dialects (MIT, LMI, and TI) was done at MIT by Richard M. Stallman. During the process of that reimplementation, some conceptual changes were made which have significantly influenced the Common Lisp Condition System.

As for the smaller but no less important proposals, Larry Masinter deserves recognition as an author of over half of them. He has worked indefatigably to write up proposals and to polish drafts by other authors. Kent Pitman, David Moon, and Sandra Loosemore have also been notably prolific, as well as Jon L White, Dan Pierson, Walter van Roggen, Skona Brittain, Scott Fahlman, and myself. Other authors of proposals include David Andre, John Aspinall, Kim Barrett, Eric Benson, Daniel Bobrow, Bob Cassels, Kathy Chapman, William Clinger, Pavel Curtis, Doug Cutting, Jeff Dalton, Linda DiMichiel, Richard Gabriel, Steven Haflich, Sonya Keene, James Kempf, Gregor Kiczales, Dieter Kolb, Barry Margolin, Chris McConnell, Jeff Peck, Jan Pedersen, Crispin Perdue, Jonathan Rees, Don Sakahara, David Touretzky, Richard Waters, and Gail Zacharias.

I am grateful to Donald E. Knuth and his colleagues for producing the TEX text formatting system [28], which was used to produce and typeset the manuscript. Knuth did an especially good job of publishing the program for TEX [29]; I had to consult the code about eight times while debugging particularly complicated macros. Thanks to the extensive indexing and cross-references, in each case it took me less than five minutes to find the relevant fragment of that 500-page program.

I also owe a debt to Leslie Lamport, author of the LATEX macro package [30] for TEX, within which I implemented the document style for this book.

Blue Sky Research sells and supports Textures, an implementation of TEX for Apple Macintosh computers; Gayla Groom and Barry Smith of Blue Sky Research provided excellent technical support when I needed it. Other software tools that were invaluable in preparing this book were QuicKeys (sold by CE Software, Inc.), which provides keyboard macros; Gōfer (sold by Microlytics, Inc.), which performs rapid text searches in multiple files; Symantec Utilities for Macintosh (sold by Symantec Corporation), which saved me from more than one disk crash; and the PostScript language and compatible fonts (sold by Adobe Systems Incorporated).

Some of this software (such as LATEX) I obtained for free and some I bought, but all have proved to be useful tools of excellent quality. I am grateful to these developers for creating them.

Electronic mail has been indispensible to the writing of this book, as well to as the work of X3J13. (It is a humbling experience to publish a book and then for the next five years to receive at least one electronic mail message a week, if not twenty, pointing out some mistake or defect.) Kudos to those develop and maintain the Internet, which arose from the Arpanet and other networks.

Chase Duffy, George Horesta, and Will Buddenhagen of Digital Press have given me much encouragement and support. David Ford designed the book and provided specifications that I could code into TEX. Alice Cheyer and Kate Schmit edited the copy for style and puzzled over the more obscure jokes with great patience. Marilyn Rowland created the index; Tim Evans and I did some polishing. Laura Fillmore and her colleagues at Editorial, Inc., have tirelessly and meticulously checked one draft after another and has kept the paperwork flowing smoothly during the last hectic weeks of proofreading, page makeup, and typesetting.

Thinking Machines Corporation has supported all my work with X3J13. I thank all my colleagues there for their encouragement and help.

Others who provided indispensible encouragement and support include Guy and Nalora Steele; David Steele; Cordon and Ruth Kerns; David, Patricia, Tavis, Jacob, Nicholas, and Daniel Auwerda; Donald and Denise Kerns; and David, Joyce, and Christine Kerns.

Большая часть книги были написана в время между 22 и 3 часами (Я не такой молодой, как раньше). Автор благодарен Барбаре, Джулии и Питеру за то, что они вынесли это, и за их любовь.

Guy L. Steele Jr.
Лексингтон, штат Массачусетс
День Всех Святых, 1989

Благодарности ПЕРВОЕ ИЗДАНИЕ (1984)

Common Lisp was designed by a diverse group of people affiliated with many institutions.

Common Lisp был разработан различными группами людей из большого количества учреждений.

Выражается благодарность участникам в разработке и реализации Common Lisp'a, а также в редактировании этой книги.

Paul Anagnostopoulos	Digital Equipment Corporation
Dan Aronson	Carnegie-Mellon University
Alan Bawden	Massachusetts Institute of Technology
Eric Benson	University of Utah, Stanford University, and Symbolics, Incorporated
Jon Bentley	Carnegie-Mellon University and Bell Laboratories
Jerry Boetje	Digital Equipment Corporation
Gary Brooks	Texas Instruments
Rodney A. Brooks	Stanford University
Gary L. Brown	Digital Equipment Corporation
Richard L. Bryan	Symbolics, Incorporated
Glenn S. Burke	Massachusetts Institute of Technology
Howard I. Cannon	Symbolics, Incorporated
George J. Carrette	Massachusetts Institute of Technology
Robert Cassels	Symbolics, Incorporated
Monica Cellio	Carnegie-Mellon University
David Dill	Carnegie-Mellon University
Scott E. Fahlman	Carnegie-Mellon University
Richard J. Fateman	University of California, Berkeley
Neal Feinberg	Carnegie-Mellon University

Ron Fischer	Rutgers University
John Foderaro	University of California, Berkeley
Steve Ford	Texas Instruments

Richard P. Gabriel	Stanford University and Lawrence Livermore National Laboratory
Joseph Ginder	Carnegie-Mellon University and Perq Systems Corp.
Bernard S. Greenberg	Symbolics, Incorporated
Richard Greenblatt	Lisp Machines Incorporated (LMI)
Martin L. Griss	University of Utah and Hewlett-Packard Incorporated
Steven Handerson	Carnegie-Mellon University
Charles L. Hedrick	Rutgers University
Gail Kaiser	Carnegie-Mellon University
Earl A. Killian	Lawrence Livermore National Laboratory
Steve Krueger	Texas Instruments
John L. Kulp	Symbolics, Incorporated
Jim Large	Carnegie-Mellon University
Rob Maclachlan	Carnegie-Mellon University
William Maddox	Carnegie-Mellon University
Larry M. Masinter	Xerox Corporation, Palo Alto Research Center
John McCarthy	Stanford University
Michael E. McMahon	Symbolics, Incorporated
Brian Milnes	Carnegie-Mellon University
David A. Moon	Symbolics, Incorporated
Beryl Morrison	Digital Equipment Corporation
Don Morrison	University of Utah
Dan Pierson	Digital Equipment Corporation
Kent M. Pitman	Massachusetts Institute of Technology
Jonathan Rees	Yale University
Walter van Roggen	Digital Equipment Corporation
Susan Rosenbaum	Texas Instruments
William L. Scherlis	Carnegie-Mellon University
Lee Schumacher	Carnegie-Mellon University
Richard M. Stallman	Massachusetts Institute of Technology
Barbara K. Steele	Carnegie-Mellon University
Guy L. Steele Jr.	Carnegie-Mellon University and Tartan Laboratories Incorporated
Peter Szolovits	Massachusetts Institute of Technology
William vanMelle	Xerox Corporation, Palo Alto Research Center
Ellen Waldrum	Texas Instruments
Allan C. Wechsler	Symbolics, Incorporated
Daniel L. Weinreb	Symbolics, Incorporated

Jon L White	Xerox Corporation, Palo Alto Research Center
Skef Wholey	Carnegie-Mellon University
Richard Zippel	Massachusetts Institute of Technology
Leonard Zubkoff	Carnegie-Mellon University and Tartan Laboratories Incorporated

Кто-то принимал относительно небольшое участие. Другие люди с большой преданностью затрачивали огромные усилия. Некоторые из участников были достойными противниками (и не обязательно одобрили окончательный проект), но их острая критика была столь же важна для совершенствования Common Lisp'a, как и все положительно сформулированные предложения. Все названные выше люди помогли в той или иной форме, и автор благодарен им за проявленный интерес и дух сотрудничества, что позволило большинству принимать решения на основе соглашения после соответствующих обсуждений.

Значительную поддержку и моральную поддержку также оказали:

Norma Abel	Digital Equipment Corporation
Roger Bate	Texas Instruments
Harvey Cragon	Texas Instruments
Dennis Duncan	Digital Equipment Corporation
Sam Fuller	Digital Equipment Corporation
A. Nico Habermann	Carnegie-Mellon University
Berthold K. P. Horn	Massachusetts Institute of Technology
Gene Kromer	Texas Instruments
Gene Matthews	Texas Instruments
Allan Newell	Carnegie-Mellon University
Dana Scott	Carnegie-Mellon University
Harry Tennant	Texas Instruments
Patrick H. Winston	Massachusetts Institute of Technology
Lowell Wood	Lawrence Livermore National Laboratory
William A. Wulf	Carnegie-Mellon University and Tartan Laboratories Incorporated

Автор очень благодарен каждому из них.

Jan Zubkoff из Carnegie-Mellon University предоставил большую помощь в организации, секретарскую поддержку и неизменно хорошее настроение перед встречей с трудностями.

Разработка Common Lisp'а не могла бы быть возможна без системы электронных сообщений, предоставленной ARPANET. Проектные решения были сделаны на основе сотен различных точек зрения в большинстве случаев путём соглашения, а также, когда необходимо, просто голосованием и преобладанием большинства. За исключением двух однодневных встреч лицом к лицу, все конструкции языка и обсуждения были сделаны с помощью системы сообщений ARPANET, которая позволяет лёгкое распространение сообщений десятков людей, и несколько изменений FIXME в день. Система сообщений также предоставляет автоматическую архивацию всего обсуждения, что оказало неоценимое значение в подготовке этого руководства. В течение тридцати месяцев было отправлено около 3000 сообщений (в среднем три в день), размерами от одной строки до двадцати страниц. Предполагая 5000 знаком в каждой странице текста, все обсуждение составило около 1100 страниц. В случае с любыми другими средствами общения, обсуждение было бы провести гораздо сложнее и это бы заняло существенно большее количество времени.

Идея Common Lisp'а пришли из большого количества источников и были усовершенствованы в длительных обсуждениях. Я несу ответственность за форму этой книги, и за любые ошибки или несоответствия, которые могли остаться. Но честь за разработку и поддержку Common Lisp'а лежит на лицах, указанных выше, каждое из которых внесло значительный вклад.

Организация и содержание этой книги были вдохновлены в значительной степени источниками *Maclisp Reference Manual* от David A. Moon и другими [33] и *LISP Machine Manual* (четвёртое издание) от Daniel Weinreb and David Moon [55], которые в свою очередь отмечают усилия Ричарда Столлмана, Майк Мак-Магон, Алан Bawden, Гленн Бурк, и «многих людей, которых слишком много, чтобы перечислить».

Автор благодарит Phyllis Keenan, Chase Duffy, Virginia Anderson, John Osborn, и Jonathan Baker из Digital Press за их помощь в подготовке этой книги для публикации. Jane Blake проделал большую работу по редактированию. James Gibson и Katherine Downs из Waldman Graphics много поработали в печати этой книги из моих исходников FIXME.

Автор благодарен Университету Карнеги-Мелона и Tartan Laboratories Incorporated за поддержку в написании этой книги в течение последних трёх лет.

Часть работы над этой книгой была сделана совместно с Carnegie-

Mellon University Project Spice, который служит для того, чтобы построить среду для передовых научных разработок программного обеспечения для персональных компьютеров. The Project Spice поддерживается с помощью Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, контролируется Air Force Avionics Laboratory under contract F33615-78-C-1551. Мнения и выводы, содержащиеся в этой книге, принадлежат автору и не должно толковаться как представляющие официальную политику, явно выраженную или подразумеваемую Defense Advanced Research Projects Agency или правительством Соединённых Штатов.

Большая часть этой книги писалась в интервале между полуночью и пятью утра. Автор благодарен Барбаре, Джулии и Питеру за то, что они вынесли это, и за их любовь.

Guy L. Steele Jr.
Лексингтон, штат Массачусетс
Март, 1984

Глава 1

Вступление

Common Lisp это новый диалект Lisp'а, наследник MacLisp'а [33, 37], под влиянием ZetaLisp'а [55, 34], в некоторой мере расширенный Schema'ой [46] и Interlisp'ом [50].

1.1 Цель

Common Lisp предназначен для достижения следующих целей:

Объединение Common Lisp создан в попытке сфокусировать работу несколько групп разработчиков, каждая из которых создавала потомка MacLisp для различных компьютеров. Это реализации начинали отличаться из-за различий в платформах: персональные компьютеры (Zetalisp, Spice Lisp), коммерческие компьютеры с разделением времени (NIL— «Новая реализация Lisp'а») и суперкомпьютеры (S-1 Lisp). Тогда как различия между платформам приводят к несовместимостям между реализациями, Common Lisp предоставляет общий диалект, который каждая реализация будет расширять для своих потребностей.

Переносимость Переносимость Common Lisp умышленно исключает функционал, который не может быть легко реализован на широком спектре машин. С одной стороны, сложный или дорогой в аппаратной реализации без специальных микрокодов функционал исключается или представляется в более абстрактной и эффективно реализуемой форме. (Примером тому являются

невидимые ссылочные указатели и локативы ZetaLisp'a. Некоторые из решаемых ими проблем в CommonLisp разрешаются другими путями.) С другой стороны, функционал полезный только на некоторых «обычных» или «коммерческих» процессорах исключается или делается опциональными. (Примером тому является система декларации типа, которая на некоторых реализациях полезна, а на других полностью игнорируется. Декларации типов полностью опциональны и в правильных программах влияют только на эффективность, а не на семантику.) Common Lisp спроектирован для упрощения построения программ, которые как можно меньше зависят от машинно-специфичных характеристик, таких, как, например, длина слова, но при этом допускает некоторые различия реализаций.

Согласованность Многие реализации Lisp'а внутренне не согласованы в том, что семантика одной и той же корректной программы может различаться для интерпретатора и компилятора. Эти семантические различия преимущественно вытекают из факта, что интерпретатор считает все переменные динамическими, тогда как компилятор считает все переменные лексическими, если иное не указано явно. Такое различие было обычной практикой в Lisp'е для достижения удобства и эффективности, но могло быть причиной скрытых, очень "тонких" ошибок. Определение Common Lisp'а исключает такие аномалии явным требованием к интерпретатору и компилятору реализовывать идентичные семантики для корректных программ настолько, насколько это возможно.

Выразительность Common Lisp собрал конструкции, которые, как показывает опыт, наиболее удобны и понятны, не только из MacLisp'а, но также из других диалектов, и языков программирования. Конструкции, оценённые как неуклюжие или бесполезные были исключены.

Совместимость Common Lisp старается быть совместимым с Lisp Machine Lisp'ом, MacLisp'ом и Interlisp'ом, примерно в таком порядке.

Эффективность Common Lisp содержит много функционала,

созданного для облегчения производства высококачественного скомпилированного кода в тех реализациях, разработчики которых заинтересованы в создании эффективного компилятора. Одна реализация Common Lisp'а называемая S-1 Lisp, уже содержит компилятор, который производит код для математических вычислений, который конкурирует в скорости выполнения с кодом, произведенных компилятором Fortran'а [11]. Компилятор S-1 Lisp дополняет по созданию наиболее эффективных численных вычислений, проделанную в MacLisp'е [19].

Мощность Common Lisp является потомком MacLisp'а, который традиционно делал акцент на предоставлении иструментов для построения систем. Такие инструменты в свою очередь могли быть использованы для создания пользовательских пакетов, вроде тех, что предоставлял Interlisp. Эти пакеты, однако, не являются частью спецификации Common Lisp'а. Ожидается, что такие пакеты будут построены на основе Common Lisp ядра.

Стабильность Предполагается, что Common Lisp будет изменяться медленно с должным обдумыванием. Различные диалекты, которые являются надмножествами Common Lisp'а, могут служить лабораториями для тестирования расширений языка, но такие расширения будут добавляться в Common Lisp только после внимательного изучения и экспериментов.

Цели Common Lisp'а, таким образом, очень близки к Standard Lisp'у [31] и Portable Standard Lisp'у [51]. Common Lisp отличается от Standard Lisp'а преимущественно тем, что содержит больше возможностей, включая более богатую и более сложную систему типов и более сложные управляющие конструкции.

Эта книга прежде всего предназначена быть спецификацией языка, а не описанием реализации (однако, примечания для реализаций встречаются в тексте). Книга определяет набор стандартных языковых концепций и конструкций, которые могут использоваться для связи данных и алгоритмов в диалекте Common Lisp. Этот набор концепций и конструкций иногда называют «ядром языка Common Lisp», потому что он содержит концептуально необходимые или важные вещи. Это ядро не является необходимым минимумом для реализации. Несмотря на то, что многие из его конструкций могут

быть определены через другие, просто написанием Lisp кода, все же кажется, что они должны быть концептуальными примитивами, чтобы было согласие между пользователями по поводу того, как ими пользоваться. (Например, bignums и рациональные числа могут быть реализованы как код, оперирующий fixnum'амами. Тем не менее, для концептуальной целостности языка важно то, что пользователи считают их примитивами, и они достаточно полезны для того, чтобы внести их в стандарт.)

По большей части данная книга описывает язык программирования, но не средства программирования. Для обращения к таким стандартным программным средствам, как компилятор, редактор, функции трассировки, и отладчик, определены несколько интерфейсов, но об их природе и функционировании сказано очень мало. Предполагается, что на основе Common Lisp'а будут построены одна или более обширных сред программирования, и они будут содержить отдельную документацию.

Теперь есть много реализаций Common Lisp'а. Некоторые были запрограммированы исследовательскими группами в университетах, а некоторые — компаниями, которые продают их в коммерческих целях, и, вокруг этих реализаций фактически вырос ряд полезных сред для программирования. Более того, все вышеуказанные цели были достигнуты, и прежде всего — переносимость. Перемещение большого количества Lisp-кода с одного компьютера на другой теперь является рутинной операцией.

1.2 Условные обозначения

Для выразительности в книге используется некоторое количество условных обозначений.

1.2.1 Десятичные числа

Все числа в данной книге представлены в десятичной системе счисления кроме мест, где система счисления указывается явно. (Конечно, десятичная система обычно и используется в работе. К несчастью, в некоторых других диалектах Lisp'а, в частности MacLisp'а, нотацией по умолчанию является восьмеричная (основание 8), вместо десятичной,

и использование десятичной системы в описании Common Lisp'а в историческом контексте слегка необычно!)

1.2.2 Nil, False и пустой список

В Common Lisp'е, как и во многих диалектах Lisp'а, символ `nil` используется для представления пустого списка и булева значения «ложь». Пустой список, конечно, может, также быть записан так: `()`; это обычно означает то же, что и `nil`. (Конечно существует возможность крайне извращённым способом в пакете переопределить значение последовательности букв `nil`, которое будет обозначать не пустой список, а другой символ с этим именем. Эта мутная возможность игнорируется в данной книге.) Эти две записи могут использоваться взаимозаменяемо настолько, насколько позволяет Lisp. В данной книге используется значение (), когда необходимо подчеркнуть использование пустого списка, и `nil`, когда обозначается булево значение «ложь». Запись `'nil` (явный знак кавычки) используется для обозначения символа. Например:

```
(defun three () 3)     ;Обозначает пустой список в параметре
(append '() '()) ⇒ ();Обозначает использование пустых списков
(not nil) ⇒ t          ;Подчёркивает использование булева значения «ложь»
(get 'nil 'color)      ;Подчёркивает использование символа
```

Любой объект данных, не являющийся `nil` преобразуется в булево значение «не ложь», которое является «истиной». Символ `t` обычно используется для обозначения «истины», когда нет более подходящего значения. Когда говорится, что функция «возвращает *ложь*» или «*ложна*» в некоторых случаях, это значит, что она возвращает `nil`. Если говорится, что функция «возвращает *истину*» или «*истинна*» в некоторых случаях, это значит, что она возвращает некоторое значение отличное от `nil`, но необязательно `t`.

1.2.3 Вычисление, Раскрытие и Равенство

Выполнение Lisp кода называется *вычисление*, так как выполнение части кода обычно возвращает некоторый объект данных, называемый

значением, созданным этим кодом. Для обозначения вычисления в примерах используется символ ⇒ . Например,

(+ 4 5) ⇒ 9

означает «результатом вычисления кода `(+ 4 5)` является (или будет, или был) `9`».

Символ → используется в примерах для обозначения раскрытия макросов. Например,

(push x v) → (setf v (cons x v))

означает «результатом раскрытия формы с макросом `(push x v)` является `(setf v (cons x v))`». Это подразумевает, что две части кода делают одно и то же действие; вторая часть кода является определением того, что делает первая часть.

Символ ≡ используется в примерах для обозначения эквивалентности (тождественности). Например,

(gcd x (gcd y z)) ≡ (gcd (gcd x y) z)

означает «значение и побочные эффекты вычисления формы `(gcd x (gcd y z))` всегда являются такими же, как и значение и побочные эффекты `(gcd (gcd x y) z)` для любых значений переменных `x`, `y` и `z`». Это подразумевает, что две части кода делают одинаковые вещи. Однако ни одна из них не определяет другую путём раскрытия макросов.

1.2.4 Ошибки

Когда в книге для некоторых возникающих ситуаций указывается, что «это ошибка», это значит:

- Некорректная Common Lisp программа должна вызывать данную ситуацию.

- Если данная ситуация случилась, побочные эффекты и результаты полностью не определены, так же как и строгое соблюдение спецификации Common Lisp'a. FIXME

- Определение этой ошибки реализация Common Lisp'a может не производить. Конечно, разработчикам рекомендуется детектирование таких ошибок, когда это необходимо.

Однако, это не говорит о том, что некоторые реализации не могут определять побочные эффекты и результаты для данных ситуаций. Смысл в том, что программа для Common Lisp'a не может быть корректной и зависеть от таких побочных эффектов и результатов.

В некоторых ситуациях, если это обозначено в книге, «*сигнализируется* ошибка», и это значит:

- Если данная ситуация случилась, будет сигнализирована ошибка (см. `error` и `cerror`).

- Корректная Common Lisp программа может полагаться на тот факт, что будет сигнализирована ошибка.

- Каждая реализация Common Lisp'a должна определять такую ошибку.

В местах, где встречаются выражения «должен» или «не должен» или «невозможен», если условие не выполняется, подразумевается «это ошибка». Например: если аргумент «должен быть символом» и аргумент не символ, тогда «это ошибка». Во всех случаях, где ошибка *сигнализируется*, всегда явно используется слово «сигнализируется (генерируется)».

1.2.5 Описания функций и других объектов

Функции, переменные, именованные константы, специальные формы и макросы описываются с помощью особого типографского формата. Таблица 1.1 показывает способ, которым документируются Common Lisp функции. Первая строка определяет имя функции, способ передачи аргументов, и тот факт, что это функция. Если функция принимает много аргументов, тогда имена аргументов могут быть разнесены на две

или три строки. Параграф, следуемый за этим стандартным заголовком, разъясняет определение и использование данной функции, а также предоставляет примеры или связанные функции.

Иногда две и более связанных функций описываются в одном комбинированном параграфе. В такой ситуации заголовки для всех функций отображаются совместно с последующим описанием.

Текущий код (включая текущие имена функций) предоставляется в данном шрифте: `(cons a b)`. Имена, встречающиеся в частях кода (метапеременные) пишутся *наклонным шрифтом*. В описании функции имена параметров предоставляются в наклонном шрифте. Слово `&optional` в списке параметров указывает на то, что все последующие аргументы являются необязательными. Значения по-умолчанию для параметров описываются далее в тексте. Список параметров может также включать `&rest`, указывающий на возможность бесконечного количества аргументов, или `&key`, указывающий на то, что могут аргументы могут приниматься по имени. (`&optional`/`&rest`/`&key` синтаксис фактически используется в определениях Common Lisp функций для этих целей).

Таблица 1.2 показывает способ, с помощью которого документируются глобальные переменные. Первая строка определяет имя переменной и тот факт, что это переменная. Все глобальные переменные Common Lisp'а имеют имена, начинающиеся и заканчивающиеся звёздочкой (asterisk).

Таблица 1.3 отображает способ, с помощью которого документируются константы. Первая строка определяет имя константы и тот факт, что это константа. (Константа является просто глобальной переменной за исключением того, что при попытке связывания это переменной с другим значением возникает ошибка.)

Таблицы 1.4 и 1.5 показывают документирование специальных форм и макросов, предназначения которых тесно связаны. Они очень сильно отличаются от функций. Функции вызываются в соответствии с одним определённым неизменным механизмом. `&optional`/`&rest`/`&key` синтаксис задаёт то, как функция внутренне использует свои аргументы, но не влияет на механизм вызова. В отличие от этого, каждая специальная форма или макрос может иметь свой особенный идиосинкразический механизм. Синтаксис Common Lisp'а задаётся и расширяется с помощью специальных форм и макросов.

В описании специальных форм или макросов, наклонные слова

обозначают соответствующую часть формы, которая вызывает специальную форму или макрос. Круглые скобки означают сами себя, и таким же образом должны быть указаны при вызове специальной формы или макроса. Квадратные скобки, фигурные скобки, звездочки, знаки плюса, и вертикальные скобки является метасинтаксическими знаками. Квадратные скобки, [и] показывают, что заключённое в них выражение является необязательным (может встречаться ноль и один раз в данном месте); квадратные скобки не должны записываться в коде. Фигурные скобки, { и }, просто отображают заключённое в них выражение, однако после закрывающей скобки может следовать звёздочка, * или знак плюс +. Звёздочка показывает, что выражение в скобках может встречаться НОЛЬ и более раз, тогда как плюс показывает, что выражение может встречаться ОДИН и более раз. Внутри скобок, может использоваться вертикальная черта |, она разделяет взаимоисключаемые элементы выбора. В целом, запись $\{x\}^*$ значит, что x может встречаться ноль и более раз, запись $\{x\}+$ значит, что x может встречаться один и более раз, и запись $[x]$ значит, что x может встречаться ноль или один раз. Такие записи также используются для описания выражений в стиле БНФ, как в таблице 22.4.

Двойные скобки, [[и]], показывают, что может использоваться любое количество альтернатив перечисленных в скобках в любом порядке, но каждая альтернатива может использоваться только один раз, если только за ней нет звёздочки. Например,

p [[*x* | {*y*}* | *z*]] *q*

означает, что, как максимум один x, любое количество y, и как максимум один z могут в любом порядке использоваться между p и q.

Стрелочка вниз, $\downarrow$, показывает, что ниже будет раскрываться данная форма. Это делает запись [[ ]] более читаемой. Если X является некоторым нетерминальным символом стоящим слева в некоторой БНФ форме, правая часть должна быть подставлена вместо символа $\downarrow X$ во всех случаях его использования. Вот два фрагмента

p [[↓*xyz-mixture*]] *q*
xyz-mixture ::= *x* | {*y*}* | *z*

вместе составляют эквивалент для предыдущего примера.

В последнем примере в таблице 1.5, рассматривается использование записи с точкой. Точка, встречающаяся в выражении `(sample-macro` *`var . body`*`)`, означает то, что имя *body* является списком форм, и не одиночной формой в конце списка. Эта запись часто используется в примерах.

В заглавной строке в таблице 1.5, запись [[ ]] означает, что может указываться любое количество деклараций, но максимум одна строка документации (которая может указываться перед, после, или в где-то в середине любого определения).

1.2.6 Лисповый считыватель

Термин «Лисповый считыватель (читатель лиспового кода)» не относится к вам, читатель этой книги, и не к какому-либо человеку читающему код на Lisp'е, а именно к Lisp процедуре, которая называется `read`. Она читает символы из входного потока и интерпретирует их с помощью парсинга, как Lisp объекты.

1.2.7 Обзор синтаксиса

В Common Lisp'е некоторые строковые символы используется в определённых целях. Полное описание синтаксиса можно прочесть в главе 22, но небольшой обзор здесь может быть также полезен:

`(` Левая круглая скобка начинает список элементов. Список может содержать любое количество элементов, включая ноль (пустой список). Списки могут быть вложенными. Например, `(cons (car x) (cdr y))` список из трёх элементов, в котором два последних также являются списками.

`)` Правая круглая скобка завершает список элементов.

`'` Одинарная кавычка или апостроф с последующим выражением *form* является сокращением для `(quote` *`form`*`)`.

`;` Точка с запятой обозначает комментарий. Она и все символы после неё до конца строки игнорируются.

" Двойная кавычка окружает символьные строки:
`"This is a thirty-nine-character string."`

\ Обратная наклонная черта является экранирующим символом. Она показывает, что следующий символ считывается как обычный, не влияющий на синтаксическую конструкцию. Например, `A\(B` означает символ, имя которого содержит три знака `A`, `(` и `B`. Подобным образом `"\""` означает, что строка содержит один знак - двойную кавычку. Первая и последняя двойные кавычки обозначают начало и конец строки. Обратная наклонная черта обозначает, что вторая двойная кавычка интерпретируется как знак, а не как синтаксическая конструкция для обозначения начала и конца строки.

| Вертикальные черты используются попарно, для окружения имени (или части имени) символа, которое содержит много специальных знаков. Это равнозначно тому, что перед каждым из них ставилась бы обратная косая черта. Например, `|A(B)|`, `A|(|B|)|` и `A\(B\)` все означают одно и то же имя.

Знак решётки (диез) обозначает начало сложной управляющей конструкции. Следующий символ определяет синтаксис следуемой конструкции. Например, `#o105` означает 105_8 (105 в восьмеричной системе счисления); `#x105` означает 105_{16} (105 в шестнадцатеричной системе счисления); `#b1011` означает 1011_2 (1011 в двоичной системе счисления); `#\L` определяет строковый символ `L`; и `#(a b c)` обозначает вектор из трёх элементов `a`, `b` и `c`. В частности важным случаем является то, что `#'`*fn* означает `(function` *fn*`)`, на манер использования одинарной кавычки `'`*form* обозначающей `(quote` *form*`)`.

‘ Обратная одинарная кавычка показывает, что следуемое выражение является шаблоном, который может содержать запятые. Синтаксис этой обратной кавычки представляет программу, которая может создавать структуры данных в соответствии с шаблоном.

, Запятые используются внутри конструкции с обратной кавычкой.

: Двоеточие используется для отображения того, к какому пакету принадлежит символ. Например, `network:reset` показывает, что символ с именем `reset` принадлежит пакету `network`. Двоеточие в начале обозначает keyword (примечания переводчика: это не ключевое слово в понимании других языков программирования), символ, который вычисляется сам в себя. Двоеточие не является частью печатаемого имени символа. Это все объясняется в главе 11; пока вы её не прочли, просто держите в голове, что символ с двоеточием в начале являлется константой, которая вычисляется сама в себя.

Квадратные и фигурные скобки, вопросительный и восклицательные знаки, (`[`, `]`, `{`, `}`, `?` и `!`) ни для каких целей не используются в стандартном Common Lisp синтаксисе. Эти символы явно зарезервированы для пользователей, преимущественно для использования в качестве *макро-символов* для пользовательских расширений синтаксиса (см. раздел 22.1.3).

Весь код в данной книге записан в нижнем регистре. Common Lisp не чуствителен к регистру текста программы. Внутри при обработке Common Lisp транслирует все имена символов в верхний регистр. С помощью `*print-case*` можно настроить регистр вывода имен символов. В данной книге, там где отображается взаимодействие пользователя и Lisp'овой системы, ввод представлен в нижнем регистре, а вывод в верхнем.

Настройка переменной `readtable-case` позволяет именам символов быть регистрозависимыми. Однако, поведение по умолчанию такое, как описано в предыдущем параграфе. В любом случае, описанные в книге имена символов внутри представлены с помощью букв в верхнем регистре.

Таблица 1.1: Образец описания функций

[Функция] **sample-function** *arg1 arg2* **&optional** *arg3 arg4*

Функция `sample-function` складывает вместе *arg1* и *arg2* и полученную сумму умножает на *arg3*. Если *arg3* не задан или равен `nil`, умножения не производится. `sample-function` затем возвращает список, в котором первый элемент содержит результат, а второй элемент равен *arg4* (который по умолчанию равен `foo`). Например:

(sample-function 3 4) ⇒ (7 foo)
(sample-function 1 2 2 'bar) ⇒ (6 bar)

В целом, `(sample-function` *`x y`*`)` ≡ `(list (+` *`x y`*`) 'foo)`.

Таблица 1.2: Образец описания переменной

[Переменная] ***sample-variable***

Переменная `*sample-variable*` задаёт, сколько раз специальная форма `sample-special-form` должна выполняться. Значение должно быть всего неотрицательным числом или `nil` (что значит, выполнение бесконечно много раз). Начальное значение `0` (означает, отсутствие выполнения).

Таблица 1.3: Образец описания константы

[Константа] **sample-constant**

Именованная константа `sample-constant` хранит значение высоты экрана терминала равное произведению одной восьмой и логарифма по основанию 2 от общего объёма диска в байтах, как число с плавающей точкой. FIXME

Таблица 1.4: Образец описания специальной формы

[Специальный оператор] **sample-special-form** [name] ({var}*) {form}+

Производит вычисление каждой формы в последовательности, как неявный `progn`, и делает это столько раз, сколько обозначено в глобальной переменной `*sample-variable*`. Каждая переменная *var* связывается и инициализируется значением `43` перед тем, как выполнить первую итерацию, и освобождается после последней итерации. Имя *name*, если задано, может быть использовано в `return-from` форме для преждевременного выхода из цикла. Если цикл завершился нормально, `sample-special-form` возвращает `nil`. Например:

(setq *sample-variable* 3)
(sample-special-form () *form1 form2*)

Здесь вычисляется *form1*, *form2*, *form1*, *form2*, *form1*, *form2* в указанном порядке.

Таблица 1.5: Образец описания макроса

[Макрос] **sample-macro** var [[declaration* | doc-string]] {tag | statement}*

Вычисляет выражения, как тело `prog` с переменной *var* связанной со значением `43`.

(sample-macro x (return (+ x x))) ⇒ 86
(sample-macro *var* . *body*) → (prog ((*var* 43)) . *body*)

Глава 2

Типы данных

Common Lisp предоставляет множество типов для объектов данных. Необходимо подчеркнуть, что в Lisp'е типизированы данные, а не переменные. Любая переменная может содержать данные любого типа. (Можно указать явно, что некоторая переменная фактически может содержать только один или конечное множество типов объектов. Однако, такая декларация может быть опущена, и программа будет выполняться корректно. Такая декларация содержит рекомендации от пользователя, и это может быть полезным при оптимизации. Смотрите `declare`.)

В Common Lisp'е тип данных является (возможно бесконечным) множеством Lisp объектов. Многие объекты Lisp'а принадлежат к более чем одному множеству типов, так что иногда не имеет смысла спрашивать тип объекта; вместо этого задаётся вопрос о принадлежности объекта к нужному типу. Предикат `typep` может использоваться для определения принадлежности объекта к заданному типу, а функция `type-of` возвращает тип, к которому принадлежит заданный объект.

Типы данных в Common Lisp сложены в иерархию (фактически в порядке убывания объёма) определённую отношениями подмножеств. Несомненно множества объектов, такие как множество чисел и множество строк заслуживают идентификаторов. Для многих этих идентификаторов используются символы (здесь и далее, слово «символ» ссылается на тип Lisp'овых объектов символ, известный также как литеральный атом). См. главу 4 подробно описывающую определения типов.

Множество все объектов определяется символом `t`. Пустой тип данных, который не содержит объектов обозначается с помощью `nil`.

Следующие категории объектов Common Lisp'а в особенности интересны: числа (numbers), знаки (characters), символы (symbols), списки (lists), массивы (arrays), структуры (structures) и функции (functions). Другие типы тоже, конечно, интересны. Некоторые из этих категорий имеют много подразделов. Так же есть стандартные типы, которые определены как объединение двух и более данных категорий. Вышеупомянутые категории, являясь типами объектов, менее «реальны» чем другие типы данных. Они просто составляют объединения типов для наглядности.

Вот краткое изложение различных Common Lisp'овых типов данных. Оставшиеся разделы данной главы рассматривают типы более детально, а также описывают нотации для объектов для каждого типа. Описание Lisp'овых функций, что оперируют объектами данных каждого типа будет даваться в следующих главах.

- *Числа* имеют различные формы и представления. Common Lisp предоставляет целочисленный (integer) тип данных: любое целое число, положительное или отрицательное ограничено размерами памяти (преимущественно равными ширине машинного слова). Также предоставляется рациональный или дробный (rational) тип данных: это отношение двух целых чисел, не являющееся целым числом. Также предоставляются числа с плавающей точкой различных интервалов и точностей. И наконец, в языке также есть комплексные числа.

- *Строковые* символы представляют печатные символы, такие как буквы или управляющие форматированием символы. Строки являются одномерными массивами символов. Common Lisp предоставляет богатое множество символов, включая способы представления различных стилей печати.

- *Символы* (иногда для ясности называемые *атомные символы (atomic symbols)*) являются именованными объектами данных. Lisp предоставляет механизм определяющий местоположение объекта символа по заданному имени (в форме строки). У символов есть *списки свойств*, которые фактически позволяют использовать символы в качестве структур, с расширяемым множеством имён

полей, каждое из которых может быть любым Lisp объектом. Символы также служат для именования функций и переменных в программе.

- *Списки* это последовательность, представленная в форме связанных ячеек, называемых *cons-ячейками*. Для обозначения пустого списка служит специальный объект (обозначаемый символом `nil`). Все остальные списки создаются рекурсивно, с помощью добавления новых элементов в начало существующего списка. Это происходит так: создаётся новая cons-ячейка, которая является объектом, имеющим два компонента, называемых *car* и *cdr*. *Car* может хранить, что угодно, а *cdr* создан для хранения указателя на существующий ранее список. (Cons-ячейки могут использоваться для хранения записи структуры из двух элементов, но это не главное их предназначение.)

- *Массивы* - это n-мерные коллекции объектов. Массив может иметь любое неотрицательное количество измерений и индексироваться с помощью последовательности целых чисел. Общий тип массива может содержать любой Lisp объект. Другие типы массивов специализируются для эффективности и могут содержать только определённые типы Lisp объектов. Также существует возможность того, что два массива, возможно с разным количеством измерений, указывают на одно и то же подмножество объектов (если изменить первый массив, изменится и второй). Это достигается с помощью указания для одного массива *быть связанным* с другим массивом. Одномерные массивы любого типа называются *векторами (vectors)*. Одномерные массивы строковых символов называются *строки*. Одномерные массивы битов (это целое число, которое может содержать 0 или 1) называются *битовыми векторами (bit-vectors)*.

- *Хеш-таблицы* предоставляют эффективный способ связывания любого Lisp объекта (*ключа*) с другим объектом (*значением*).

- *Таблицы символов Lisp парсера (readtables)* используются для управления парсером выражений `read`. Этот функционал предназначен для создания макроридеров для ограниченного изменения синтаксиса языка.

- *Пакеты* являются коллекциями символов и служат для разделения их на пространства имён. Парсер распознает символы с помощью поиска их имён в текущем пакете.

- *Имена файлов (pathnames)* хранят в себе путь к файлу в кроссплатформенном виде. Они используются для взаимодействия с внешней файловой системой.

- *Потоки* представляют источники данных, обычно строковых символов или байтов. Они используются для ввода/вывода, а также для внутренних нужд, например для парсинга строк.

- *Состояния генератора случайных чисел (random-states)* — это структуры данных, используемые для хранения состояния встроенного генератора случайных чисел (ГСЧ).

- *Структуры* — это определённые пользователем объекты, имеющие именованные поля. `defstruct` используется для определения новых типов структур. Некоторые реализации Common Lisp'a могут внутренне предоставлять некоторые системные типы такие, как *bignums*, *таблицы символов Lisp парсера (readtables)*, *потоки (streams)*, *хеш-таблицы (hash tables)* и *имена файлов (pathnames)* как структуры, но для пользователя это не имеет значения.

- *Условия (conditions)* — это объекты, используемые для управления ходом выполнения программы с помощью сигналов и обработчиков этих самых сигналов. В частности, ошибки сигнализируются с помощью генерации условия, и эти ошибки могут быть обработаны с помощью установленных для этих условий обработчиков.

- *Классы* определяют структуру и поведение других объектов, являющихся *экземплярами* данных классов. Каждый объект данных принадлежит некоторому классу.

- *Методы* — это код, который оперирует аргументами, которые соответствуют некоторому шаблону. Методы не являются функциями; они не вызываются напрямую, а объединяются в обобщённые функции (generic functions).

- *Обобщённые функции* — это функции, которые содержат, кроме всего прочего, множество методов. При вызове generic

функция вызывает подмножество её методов. Подмножество для выполнения выделяется с помощью определения классов аргументов и выбора им соответствующих методов.

Эти категории не всегда взаимоисключаемы. Указанные отношения между различными типами данных более детально описано в разделе 2.15.

2.1 Числа

В Common Lisp'е определены некоторые виды чисел. Они подразделяются на *целочисленные (`integer`)*; *дробные (`ratio`)*; *с плавающей точкой (floating-point)* с четырьмя видами представления, *действительные (`real`)* и *комплексные (`complex`)*.

Числовой (`number`) тип данных захватывает все числовые типы. Для удобства также предоставлены имена для некоторых числовых подтипов. Целые числа и дроби принадлежат к *рациональному (`rational`)*. Рациональные числа и с плавающей точкой — к *действительному (`real`)*. Действительные (real) и комплексные (complex) — к *числовому (`number`)* типу.

Несмотря на то, что эти типы выбирались из математической терминологии, соответствие не всегда полное. Модель целочисленных (integers) и дробных (ratios) типов полностью совпадает с математической. Числа с *плавающей точкой (float)* могут использоваться для аппроксимации действительных (real) чисел: рациональных (rational) и иррациональных (irrational). *Действительный (real)* тип включает все Common Lisp числа, которые отображают действительные (real) математические числа, однако для математических иррациональных (irrational) аналогии в Common Lisp'е нет. Только *действительные (real)* числа могут быть отсортированы с помощью функций <, >, <= и >=. (Ох жеш FIXME).

2.1.1 Целые числа

Целочисленный тип данных предназначен для отображения математических целых чисел. В отличие от большинства языков программирования, Common Lisp принципиально не навязывает

ограничений на величину целого числа. Место для хранения больших чисел выделяется автоматически по мере необходимости.

В каждой реализации Common Lisp'а есть интервал целых чисел, которые хранятся более оптимально, чем другие. Каждое такое число называется *fixnum*, и число не являющееся fixnum'ом называется *bignum*. Common Lisp спроектирован так, чтобы скрыть различие настолько, насколько это возможно. Различие между fixnums и bignums видимо пользователю, только в тех местах, где важна эффективность работы алгоритма. Какие числа являются fixnums зависит от реализации; обычно это числа в интервале от -2^n to $2^n - 1$, включительно, для некоторого n не меньше 15. См. `most-positive-fixnum` и `most-negative-fixnum`.

`fixnum` должен быть супертипом для типа `(signed-byte 16)`, и в дополнение к этому, значения `array-dimension-limit` должны принадлежать fixnum (разработчики должны выбрать интервал fixnum, чтобы в него можно было включить наибольшее число поддерживаемых измерений для массивов).

Обоснование: Эта спецификация позволяет программистам объявлять переменные в переносимом коде типа `fixnum` для эффективности. Fixnums гарантированно заключают в себе множество знаковых 16-битных чисел чисел (это сравнимо с типом данных `short int` в языке программирования C). В дополнение к всему, любой корректный индекс массива должен быть fixnum, и в таком случае переменные, которые хранят индексы массива (например переменная в `dotimes`) могут быть объявлены как `fixnum` в переносимом коде.

Целые числа обычно записываются в десятичном виде, как последовательность десятичных цифр опционально с предшествующим знаком и опционально с последующей точкой. Например:

```
                0     ;Нуль
               -0     ;Это всегда значит то же, что и 0
               +6     ;Первое совершенное число
               28     ;Второе совершенное число
             1024.    ;Два в десятой степени
               -1     ;e^(πi)
15511210043330985984000000.  ;факториал от 25 (25!), скорее все bignum
```

Целые числа могут быть представлены с основаниями отличными от десяти. Синтаксис:

#*nn*r*ddddd* or #*nn*R*ddddd*

означает, что целое число с основанием *nn* определённое с помощью цифр и букв *ddddd*. Более точное описание: символ `#`, не пустая последовательность десятичных цифр представляющих десятичное число *n*, `r` (или `R`), опционально знак + или -, и последовательность цифр для заданной системы счисления (система счисления должна быть между 2 и 36, включительно). Для заданной системы счисления могут использоваться для задания числа только корректные символы. Например для восьмеричного числа могут использоваться только цифры от 0 до 7 включительно. Для систем счисления больших десятичной, могут использоваться буквы алфавита в любом регистре в алфавитном порядке. Двоичные, восьмеричные и шестнадцатеричные основания можно использовать с помощью следующих аббревиатур: `#b` для `#2r`, `#o` для `#8r`, `#x` для `#16r`. Например:

```
#2r11010101     ;Другой способ определения числа 213
 #b11010101     ;то же самое
#b+11010101     ;то же самое
   #o235        ;То же самое, в восьмеричной системе
    #xD5        ;То же самое, в шестнадцатеричной системе
  #16r+D5       ;То же самое
   #o-300       ;Десятичное число -192, записанное восьмеричным числом
  #3r-21010     ;То же самое, в троичное системе счисления
   #25R-7H      ;То же самое с основанием 25
#xACCEDED       ;181202413, в шестнадцатеричной системе
```

2.1.2 Дробные числа

Дробное число — это число отображающее математическое отношение между двумя целыми числами. Целые и дробные числа вместе составляют тип рациональных (`rational`) чисел. Образцовое

отображение дробных чисел - это целое число, если значение целое, в противном случае это отношение двух целых чисел, *числителя* и *знаменателя*, наибольший общий делитель которых единица, в котором знаменатель положителен (и фактически больший чем единица, иначе дробь является целым числом). Дробь записывается с помощью разделителя `/`, так: `3/5`. Есть возможность использовать нестандартную запись такую, как `4/6`, но Lisp функция `prin1` всегда выводит дробь в стандартной форме.

Если какое-либо вычисление привело к результату, являющемуся дробью двух целых чисел, где знаменатель делит числитель нацело, тогда результат немедленно преобразуется в эквивалентное целое число. Это называется правилом *канонизации дробей*.

Дробные числа могут быть записаны так: опционально знак + или -, за ним следуют две не пустые последовательности цифр разделённых с помощью `/` . Такой синтаксис может быть описан так:

ratio ::= [sign] {digit}+ `/` {digit}+

Вторая последовательность не может состоять только из нулей. Например:

```
2/3                 ;Это каноническая запись
4/6                 ;Это неканоническая запись предыдущего числа
-17/23              ;Не очень интересная дробь
-30517578125/32768  ;Это (−5/12)^15
10/5                ;Это каноническая запись для 2
```

Для задания дробей в системе счисления отличной от десятичной, необходимо использовать спецификатор основания (один из `#`*nn*`R`, `#O`, `#B` или `#X`) как и для целых чисел. Например:

```
#o-101/75          ;Восьмеричная запись для -65/61
#3r120/21          ;Третичная запись для 15/7
#Xbc/ad            ;Шестнадцатеричная запись для 188/173
#xFADED/FACADE     ;Шестнадцатеричная запись для 1027565/16435934
```

2.1.3 Числа с плавающей точкой

Common Lisp позволяет реализации содержать один и более типов чисел с плавающей точкой, которые все вместе составляют тип `float`. Число с плавающей точкой является (математически) рациональным числом формы $s \cdot f \cdot b^{e-p}$, где s $+1$ или -1, является *знаком*; b целое число большее 1, является *основанием* для представления; p положительное целое, является *точностью* (количество цифр по основанию b) числа с плавающей точкой; f положительное целое между b^{p-1} и $b^p - 1$ (включительно), является мантиссой; и e целое число, является экспонентой. Значение p и интервал e зависит от реализации, также может быть «минус ноль». Если «минус ноль» отсутствует, тогда `0.0` и `-0.0` оба интерпретируются, как ноль с плавающей точкой.

Заметка для реализации: The form of the above description should not be construed to require the internal representation to be in sign-magnitude form. Two's-complement and other representations are also acceptable. Note that the radix of the internal representation may be other than 2, as on the IBM 360 and 370, which use radix 16; see `float-radix`.

В зависимости от реализации числа с плавающей точкой могут предоставляться с различными точностями и размерами. Высококачественные программы с вычислениями с плавающей точкой зависят от того, какая предоставлена точность, и не всегда могут быть полностью переносимы. Для содействия по умеренной переносимости программ, сделаны следующие определения:

- *Короткий* тип числа с плавающей точкой (тип `short-float`) является представлением числа с наименьшей фиксированной точностью, предоставляемого реализацией.

- *Длинный* тип числа с плавающей точкой (тип `long-float`) является представлением числа с наибольшей фиксированной точностью, предоставляемого реализацией.

- Промежуточными форматами между коротким и длинным форматами является два других формата, называемых *одинарный* и *двойной* (типы `single-float` и `double-float`).

Таблица 2.1: Рекомендуемый размер для точности и экспоненты для типа с плавающей точкой

Формат	Минимальная точность	Минимальный размер экспоненты
Короткое	13 бит	5 бит
Одинарное	24 бит	8 бит
Двойное	50 бит	8 бит
Длинное	50 бит	8 бит

Определение точности для этих категорий зависит от реализации. Однако, примерная цель такая, что короткий тип с плавающий точкой должен содержать точность как минимум 4 позиции после запятой (и также должен иметь эффективное представление в памяти); одинарный тип с плавающей точкой — как минимум 7 знаков после запятой; двойной тип с плавающей точкой — как минимум 14 знаков после запятой. Предполагается, что размер точности (измеряется в битах и рассчитывается как $p \log_2 b$) и экспоненты (измеряется в битах и рассчитывается как логарифм с основанием 2 от (1 плюс максимальное значение экспоненты) должен быть как минимум таким же большим как значения из таблицы 2.1.

Числа с плавающей точкой записываются в двух формах десятичной дробью и компьютеризированной научной записью: необязательный знак, затем не пустая последовательность цифр с встроенной точкой, затем необязательная часть определения экспоненты. Если определения экспоненты нет, тогда требуется точка, и после неё должны быть цифры. Определение экспоненты составляется из маркера экспоненты, необязательного знака и негустой последовательности цифр. Для ясности приведена БНФ для записи чисел с плавающей точкой.

число-с-плавающей-точкой ::= [знак] {цифра}* *точка* {цифра}+ [экспонента]
| [*знак*] {цифра}+ [точка {цифра}*] *экспонента*
знак ::= `+` | `-`
точка ::= `.`
цифра ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
экспонента ::= *маркер-экспоненты* [знак] {цифра}+
маркер-экспоненты ::= `e` | `s` | `f` | `d` | `l` | `E` | `S` | `F` | `D` | `L`

Если определение экспоненты отсутствует или если используется

маркер экспоненты `e` (или `E`), тогда используемые формат точности не задан. Когда такое представление считывается и конвертируется во внутренний формат объекта числа с плавающей точкой, формат задаётся с помощью переменной `*read-default-float-format*`; первоначальное значение данной переменной `single-float`.

Буквы `s`, `f`, `d` и `l` (или их эквиваленты в верхнем регистре) явно задают использование типа: *короткий*, *одинарный*, *двойной* и *длинный*, соответственно.

Примеры чисел с плавающей точкой:

```
0.0                  ;Ноль с плавающей точкой в формате по умолчанию
0E0                  ;Также ноль с плавающей точкой в формате по умолчанию
-.0                  ;Это может быть нулём или минус нулём
                     ; в зависимости от реализации
0.                   ;Целый ноль, не с плавающей точкой!
0.0s0                ;Ноль с плавающей точкой в коротком формате
0s0                  ;Также ноль с плавающей точкой в коротком формате
3.14159265358979323846d0     ;Аппроксимация числе пи в двойном формате
6.02E+23             ;Число Авогадро в формате по умолчанию
602E+21              ;Также число Авогадро в формате по умолчанию
3.010299957f-1       ;log10 2, в одинарном формате
-0.000000001s9       ;e^(πi) в коротком формате
```

Внутренний формат, используемый для внешнего представления, зависит только от маркера экспоненты и не зависит от количества знаков после запятой во внешнем представлении.

Тогда как Common Lisp содержит терминологию и систему обозначений для включения 4 различных типов чисел с плавающей точкой, не все реализации будут иметь желание поддержки такого большого количества типов. Реализации разрешается предоставлять меньшее, чем 4, количество внутренних форматов чисел с плавающей точкой, в таком случае как минимум один из этих типов будет «общим» для более одного внешнего имени *короткого*, *одинарного*, *двойного* и *длинного* в соответствии со следующими правилами:

- Если предоставляется один внутренний формат, то он рассматривается как *одинарный*, но также служит *коротким*, *двойным* и *длинным*. Типы данный `short-float`, `single-float`,

`double-float` и `long-float` считаются идентичными. В этой реализации такое выражение, как `(eql 1.0s0 1.0d0)` будет истинным, потому что два числа `1.0s0` и `1.0d0` будут конвертированы в один и тот же внутренний формат и таким образом будут считаться принадлежащим одному типу данных, несмотря на различный внешний синтаксис. В такой реализации и выражение `(typep 1.0L0 'short-float)` будет истинным. В механизме вывода все числа с плавающей точкой будут выводится в *одинарном* формате, таким образом будет вывод экспоненты с буквой `E` или `F`.

- Если предоставляется два внутренних формата, то может быть выбрано одно из двух соответствий, в зависимости от того, какое является более подходящим:
  - Один формат является *коротким*, другой *длинным*, и они представлены и как *двойной* и *длинный*. Типы данных `single-float`, `double-float` и `long-float` считаются идентичными, но `short-float` от них отличается. Выражение, такое как `(eql 1.0s0 1.0d0)` будет ложным, но `(eql 1.0f0 1.0d0)` будет истинным. Также и `(typep 1.0L0 'short-float)` будет ложным, но `(typep 1.0L0 'single-float)` будет истинным. В механизме вывода все числа с плавающей точкой считаются *короткими* или *одинарными*.
  - Одни формат *одинарный* и также представляет *короткий*. Другой формат *двойной* и также представляет *длинный*. Типы данных `short-float` и `single-float` считаются одинаковыми, и `double-float` и `long-float` считаются одинаковыми. Такое выражение, как `(eql 1.0s0 1.0d0)` будет ложным, так же как и `(eql 1.0f0 1.0d0)`, но `(eql 1.0d0 1.0L0)` будет положительным. Также `(typep 1.0L0 'short-float)` будет ложным, но `(typep 1.0L0 'double-float)` будет истинным. В механизме вывода все числа с плавающей точкой считаются *одинарными* или *двойными*.
- Если предоставляется три внутренних формата, тогда может быть выбрано одно из двух соответствий, в зависимости от того, какое является более подходящим:

– Один формат *короткий*, другой *одинарный* и третий *двойной* и также рассматривается как *длинный*. Тогда применяются уже названные ограничения.

– Один формат *одинарный* и также рассматривается, как *короткий*, другой формат *двойной* и третий *длинный*.

Заметка для реализации: Рекомендуется предоставлять столько различных типов чисел с плавающей точкой, сколько возможно, используя при этом таблицу 2.1 в качестве указаний. В идеальной ситуации, *короткий* формат числа с плавающей точкой должен быть «быстрым», в частности, не требующим выделения места в куче. *одинарный* должен приближаться к стандарту IEEE для одинарного формата. *двойной* должен приближаться к стандарту IEEE для двойного формата. [23, 17, 16].

2.1.4 Комплексные числа

Комплексные числа (тип `complex`) представляются в алгебраической форме, с действительной и мнимой частями, каждая из которых является не комплексным числом (целым, дробным, или с плавающей точкой). Следует отметить, что части комплексного числа не обязательно числа с плавающей точкой; в это Common Lisp похож на PL/I и отличается от Fortran'a. Однако обе части должны быть одного типа: обе рациональные, или обе какого-либо формата с плавающей точкой.

Комплексные числа могут быть обозначены с помощью записи символа `#C` с последующим списком действительной и мнимой частей. Если две части, как было отмечено, не принадлежат одному типу, тогда они будут преобразованы в соответствие с правилами преобразования чисел с плавающей точкой описанными в главе 12.

```
#C(3.0s1 2.0s-1)    ;Действительная и мнимая части в коротком формате
#C(5 -3)            ;Целое Гаусса
#C(5/3 7.0)         ;Будет преобразовано в #C(1.66666 7.0)
#C(0 1)             ;Мнимая единица, i
```

Тип заданного комплексного числа определяется с помощью списка: слова `complex` и типа компонентов; например, специализированное представление для комплексных чисел с частями принадлежащими типу «короткое с плавающей точкой», будет выглядеть так `(complex short-float)`. Тип `complex` включает все представления комплексных типов.

Комплексное число типа `(complex rational)`, в котором части принадлежат рациональному типу, никогда не может содержать нулевую мнимую часть. Если в результате вычислений получится комплексное число с нулевой мнимой частью, то данное число будет автоматически конвертировано в не комплексное дробное число, равное действительной часть исходного числа. Это называется правилом *канонизации комплексного числа.* Данное правило не применяется для комплексных чисел с плавающими точками, то есть `#C(5.0 0.0)` и `5.0` различные числа.

2.2 Строковые символы

Строковые символы представляют собой объекты данных, принадлежащих типу `строковый символ (character)`.

Объект строкового символа может быть записан, как знак `#\` и последующий строковый символ. Например: `#\g` обозначает строковый символ g в нижнем регистре. Это работает достаточно хорошо для вывода символов. Невыводимые строковые символы имеют имена, и могут быть записаны с помощью `#\` и последующего имени; например, `#\Space` (или `#\SPACE` или `#\space` или `#\sPaCE`) обозначает символ пробела. Синтаксис для записи имени строкового символа после `#\`, такой же как и для Lisp символов. Однако в работе могут использоваться только те имена, которые известны данной реализации.

2.2.1 Стандартные строковые символы

Common Lisp определяет множество стандартных символов (подтип `standard-char`) для двух целей. Common Lisp программы, которые *записаны* используя множество стандартных символов, могут быть

прочитаны любой реализацией Common Lisp; и Common Lisp программы, которые *используют* только стандартные символы в качестве объектов данных, скорее всего будут портируемыми. Множество строковых символов Common Lisp состоит из символа пробела, `#\Space`, символа новой строки `#\Newline`, и следующих сорока четырёх печатаемых символов и их эквивалентов:

! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\ ] ^ _
‘ a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~

Множество стандартных строковых символов Common Lisp'а явно соответствует множеству из сорока пяти стандартных ASCII печатаемых символов и символа новой строки. Как бы то ни было, Common Lisp спроектирован так, чтобы быть независимым от ASCII кодировки символов. Например, сортировка последовательности не определена, кроме того, что можно сказать, что цифры могут быть корректно отсортированы, буквы в верхнем регистре могут быть корректно отсортированы и буквы в нижнем регистре могут быть корректно отсортированы (смотрите спецификацию функции `char<`). Другие реализация кодировка строковых символов, в частности EBCDIC, должна быть легко приспособлена (с необходимым соответствием выводимых символов).

Из сорока четырёх печатаемых символов, следующие используются с ограничениями связанными с синтаксисом Common Lisp программ:

[ ] { } ? ! ^ _ ~ $ %

Следующие строковые символы называются *слегка стандартизированными*:

#\Backspace #\Tab #\Linefeed #\Page #\Return #\Rubout

Не все реализации Common Lisp'а нуждаются в поддержке этих символов; но те реализации, что используют ASCII кодировку должны их поддерживать, соответственно BS (восьмеричный код 010), HT (011), LF (012), FF (014), CR (015) и DEL (177). Эти строковые символы не являются членами подтипа `standard-char`, если не будут созданы синонимы для них. Например, разработчик реализации может определить `#\Linefeed` или `#\Return` как синоним для `#\Newline`, или `#\Tab` как синоним для `#\Space`.

2.2.2 Разделители строк

Обработка разделителей строк является одним из самых сложных моментов в проектировании переносимой программы, преимущественно потому, что между операционными системами очень мало соглашений по этому поводу. Некоторые используют только один символ, и рекомендуемый для этого ASCII символ является символом перевода строки LF (также называемый символом новой строки, NL), но некоторые системы используют символ перевода каретки CR. Более широко используется последовательность из двух символов CR и последующем LF. Часто разделители строк не имеют выводимого представления, но неявно влияют на структурирование файла в записи, каждая запись содержит строку текста. Например, дека перфокарт имеет такую структуру.

Common Lisp предоставляет абстрактный интерфейс, требуя наличия одного символа `#\Newline`, который являет разделителем строк. (Язык C имеет подобное требование.) Реализация Common Lisp'а должна транслировать это односимвольное представление разделители в то, что требуется во внешних системах в данной операционной системе.

Заметка для реализации: How the character called `#\Newline` is represented internally is not specified here, but it is strongly suggested that the ASCII LF character be used in Common Lisp implementations that use the ASCII character encoding. The ASCII CR character is a workable, but in most cases inferior, alternative.

Требование того, что разделитель строк должен быть представлен одним символом, имеет следующие последствия. Строковый объект, записанный в середине программы и содержащий несколько строк,

должен содержать только один символ для каждого разделителя. Рассмотрим фрагмент следующего кода:

```
(setq a-string "This string
contains
forty-two characters.")
```

Между `g` and `c` должен быть только один символ, `#\Newline`; последовательность из двух строковых символов такая, как `#\Return` и`#\Newline`, некорректна. Такая же ситуация и между `s` и `f`.

Когда строковый символ `#\Newline` записывается в выходной файл, реализация Common Lisp'а должна предпринять соответствующие действия для разделения строк. Это может быть реализовано, как трансляция `#\Newline` в последовательность CR/LF.

Заметка для реализации: If an implementation uses the ASCII character encoding, uses the CR/LF sequence externally to delimit lines, uses LF to represent `#\Newline` internally, and supports `#\Return` as a data object corresponding to the ASCII character CR, the question arises as to what action to take when the program writes out `#\Return` followed by `#\Newline`. It should first be noted that `#\Return` is not a standard Common Lisp character, and the action to be taken when `#\Return` is written out is therefore not defined by the Common Lisp language. A plausible approach is to buffer the `#\Return` character and suppress it if and only if the next character is `#\Newline` (the net effect is to generate a CR/LF sequence). Another plausible approach is simply to ignore the difficulty and declare that writing `#\Return` and then `#\Newline` results in the sequence CR/CR/LF in the output.

2.2.3 Нестандартные символы

Любая реализация может предоставлять дополнительные строковые символы, и печатаемые и именованные. Некоторые вероятные примеры:

$\#\backslash\pi$ $\#\backslash\alpha$ #\Break #\Home-Up #\Escape

Использование таких символов, может создавать проблемы для портируемости Common Lisp программы.

2.3 Символы

Символы (примечание переводчика: не строковые) являются Lisp'овыми объектами данных, созданы для нескольких целей и имеют несколько интересных характеристик. Каждый объект типа `symbol` имеет имя, называемое его *выводимым именем (print name)*. Существует возможность получить имя символа в виде строки. Также возможно обратное действие, получение имени символа из строки. (Более подробно: символы могут быть организованы в *пакеты*, и все символы в пакете имеют уникальные имена. Смотрите главу 11.)

У символов есть компонент, называемый *список свойств*, или *plist*. Список свойств всегда является списком, у которого чётные элементы (начиная с нулевого) являются символами, они выступают в качестве имён свойств, и нечётные элементы являются связанными со свойствами значениями. Для манипуляций с этим списком свойств предоставляются функции, это позволяет символу выступать в роли расширяемой структуры.

Символы также используются для представления определённых видов переменных в Lisp программах, и для манипуляции значениями связанными с символами в такой роли также предоставляются функции.

Символ может быть обозначен просто записью его имени. Если его имя непустое, и если его имя содержит только алфавитные буквы в верхнем регистре, цифры или некоторые псевдо-алфавитные строковые символы (но не разделители, как круглые скобки и пробелы), и если имя символа не может быть интерпретировано как число, тогда имя символа задаётся последовательностью букв его имени. Во внутреннем представлении все буквы записанные в имени символа переводятся в верхний регистр. Например:

```
FROBBOZ      ;Символ, имя которого FROBBOZ
frobboz      ;Другой путь записи того же символа
fRObBoz      ;Ещё один путь записи полюбившегося символа
unwind-protect;Символ с дефисом в имени
+$           ;Символ с именем +$
```

```
1+                ;Символ с именем 1+
+1                ;Это число 1, а не символ
pascal_style      ;Этот символ содержит знак подчёркивания в своём имени
b^2-4*a*c         ;Это один символ
                  ; Этот символ содержит некоторые специальные знаки в своём имени
file.rel.43       ;Символ содержит точки в своём имени
/usr/games/zork   ;Символ содержит наклонные черты в своём имени
```

В дополнение к буквам и числам, следующие строковые символы допускаются в использовании в написании имени символа:

\+ - * / @ $ % ^ & _ = < > ~ .

Некоторые из этих строковых символов имеют специальные общепринятые значения для имён. Например, символы, которые задают специальные переменные, обычно имеют имена начинающиеся и заканчивающиеся звёздочкой `*`. Одиночная точка используется для задания cons-ячеек или списков с точкой. Точка также является разделителем дробной части.

Следующие строковые символы предназначены для использования в качестве макросимволов для изменения и расширения синтаксиса языка:

? ! [ ] { }

Выводимое имя символа может содержать буквы в верхнем и нижнем регистрах. Однако, при чтении Lisp reader обычно конвертирует буквы нижнего регистра в верхний. В реализации все символы, которые именуют все стандартные Common Lisp переменные и функции хранятся в верхнем регистре. Однако в книге все эти символы для удобства приводятся в нижнем регистре. Использование имён символов в нижнем регистре при написании программы возможно, потому что `read` конвертирует все считываемые символы в верхний регистр.

Существует функция `readtable-case`, которая контролирует поведение функции `read` касаемо преобразования регистров букв в именах символов.

Если символ не может быть задан, потому что в его имени используются недопустимые буквы и знаки, их можно «экранировать» двумя способами. Один из них заключается в использовании обратной наклонной черты перед каждым экранируемым знаком. В таком случае имя символа никогда не будет ошибочно интерпретировано, как число. Например:

```
\(                ;Символ с именем (
\+1               ;Символ с именем +1
+\1               ;Также символ с именем +1
\frobboz          ;Символ с именем fROBBOZ
3.14159265\s0     ;Символ с именем 3.14159265s0
3.14159265\S0     ;Другой символ с именем 3.14159265S0
3.14159265s0      ;short-format с плавающей точкой для аппроксимации числа π
APL\\360          ;Символ с именем APL\360
apl\\360          ;Также символ с именем APL\360
\(b^2\)\ -\ 4*a*c;Имя (B^2) - 4*A*C;
                  ; содержит скобки и два пробела
\(\b^2\)\ -\ 4*\a*c;Имя (b^2) - 4*a*c;
                  ; буквы явно указаны в нижнем регистре
```

Если таких «запрещённых» букв в имени много, использование \ перед *каждой* буквой утомительно. Альтернативным методом экранирования знаков в имени символа является заключение всего имени или только его части в скобки из вертикальных черт. Это эквивалентно тому, что каждая буква была бы экранирована обратной косой чертой.

```
|"|             ;То же что и \"
|(b^2) - 4*a*c| ;Имя (b^2) - 4*a*c
|frobboz|       ;Имя frobboz, а не FROBBOZ
|APL\360|       ;Имя APL360, потому что \ экранирует the 3
|APL\\360|      ;Имя APL\360
|apl\\360|      ;Имя apl\360
|\|\||          ;То же, что и \|\|: имя ||
|(B^2) - 4*A*C;Имя (B^2) - 4*A*C;
                  ; содержит скобки и два пробела
|(b^2) - 4*a*c| ;Имя (b^2) - 4*a*c
```

2.4 Списки и Cons-ячейки

`cons-ячейка` является записью структуры, содержащей два элемента, называемых *car* и *cdr*. Cons-ячейки используются преимущественно для отображения списков.

Список рекурсивно определяется пустым списком или cons-ячейкой, у которой *cdr* элемент является списком. Таким образом, список является цепочкой cons-ячеек связанных с помощью их *cdr* элементов, заканчивающейся пустым списком с помощью `nil`. *car* элементы cons-ячеек называются *элементами* списка. Для каждого элемента списка существует cons-ячейка. Пустой список не имеет элементов вообще.

Список записывается с помощью элементов в необходимом порядке, разделяемых пробелом (пробел, таб, возврат каретки) и окружённых круглыми скобками.

```
(a b c)              ;Список трёх элементов
(2.0s0 (a 1) #\*)    ;Список трёх элементов: короткого с плавающей точкой
                     ; числа, другого списка, и строкового символа
```

Таким образом, пустой список `nil` может быть записан, как (), потому что является списком без элементов.

Список с точкой является списком, последняя cons-ячейка которого в *cdr* элементе содержит объект данных, а не `nil` (который не является cons-ячейкой, иначе исходная cons-ячейка не была бы последней). Такой список называется «списком с точкой» по причине используемой для него специальной записи: элементы списка записанные в двух последних позициях списка перед закрывающей круглой скобкой разделяются точкой (обрамленной с двух сторон пробелами). Тогда последнее значение будет содержаться в *cdr* элементе последней cons-ячейки. В особых случаях, одиночная cons-ячейка может быть записана с помощью *car* и *cdr* элементов, обрамленных в круглые скобки и разделённых с помощью точки, окружённой пробелами. Например:

```
(a . 4)          ;cons-ячейка, car которой является символом
                 ; и cdr которой равен целому числу
```

```
(a b c . d)     ;Список с точкой с тремя элементами, у последней
                ; cons-ячейки cdr равен символу d
```

Правильной записью также является что-то наподобие `(a b . (c d))`; она означает то же, что и `(a b c d)`. Стандартный Lisp вывод никогда не распечатает список в первом виде, таким образом когда это возможно, он старается избавиться от записи с точкой.

Часто термин *список* употребляется и для обычных списков и для списков с точкой. Когда разница важна, для списка, заканчивающегося с помощью `nil`, будет употребляться термин «Ъ список». Большинство функций указывают, что оперируют списками, ожидая, что они Ъ. Везде в этой книге, если не указано иное, передача списка с точкой в такие функции является ошибкой.

Заметка для реализации: Implementors are encouraged to use the equivalent of the predicate `endp` wherever it is necessary to test for the end of a list. Whenever feasible, this test should explicitly signal an error if a list is found to be terminated by a non-`nil` atom. However, such an explicit error signal is not required, because some such tests occur in important loops where efficiency is important. In such cases, the predicate `atom` may be used to test for the end of the list, quietly treating any non-`nil` list-terminating atom as if it were `nil`.

Иногда используется термин *дерево* для ссылки на некоторую cons-ячейку, которая содержит другие cons-ячейки в своих *car* и *cdr* элементах, которые также содержат cons-ячейки в своих элементах и так далее, пока не будут достигнуты элементы, не являющиеся cons-ячейками. Такие элементы, не являющиеся cons-ячейками называются *листьями* дерева.

Списки, списки с точкой и деревья, все вместе не завершают список типов данных, они просто являются удобной точкой для рассмотрения таких структур, как cons-ячейки. Существуют также другие термины, такие как, например, *ассоциативный список*. Ни один из этих типов данных не является Lisp'овым типом данных. Типом данных являются cons-ячейки, а также `nil` является объектом типа `null`. Lisp'овый тип данных `список` подразумевает объединение типов `cons-ячеек` и `null`, и по этой причине содержит в себе оба типа: Ъ список и список с точкой.

2.5 Массивы

`Массив (array)` является объектом с элементами расположенными в соответствие с декартовой системой координат.

Количество измерений массива называется *ранг* (это терминология взята из APL). Ранг является неотрицательных целым. Также каждое измерение само по себе является неотрицательным целым. Общее количество элементов в массиве является произведением размеров всех измерений.

Реализация Common Lisp'а может налагать ограничение на ранг массива, но данное ограничение не может быть менее 7. Таким образом, любая Common Lisp программа может использовать массивы с семью и менее измерениями. (Программа может получить текущее ограничение для ранга для используемой системы с помощью константы `array-rank-limit`.)

Допускается существование нулевого ранга. В этом случае, массив не содержит элементов, и любой доступ к элементам является ошибкой. При этом другие свойства массива использоваться могут. Если ранг равен нулю, тогда массив не имеет измерений, и их произведение приравнивается к 1 (FIXME). Таким образом массив с нулевым рангом содержит один элемент.

Элемент массива задаётся последовательностью индексов. Длина данной последовательности должна равняется рангу массива. Каждый индекс должен быть неотрицательным целым строго меньшим размеру соответствующего измерения. Также индексация массива начинается с нуля, а не с единицы, как в по умолчанию Fortran'е.

В качестве примера, предположим, что переменная `foo` обозначает двумерный массив с размерами измерений 3 и 5. Первый индекс может быть 0, 1 или 2, и второй индекс может быть 0, 1, 2, 3 или 4. Обращение к элементам массива может быть осуществлено с помощью функции `aref`, например, `(aref foo 2 1)` ссылается на элемент массива (2, 1). Следует отметить, что `aref` принимает переменное число аргументов: массив, и столько индексов, сколько измерений у массива. Массив с нулевым рангом не имеет измерений, и в таком случае `aref` принимает только один параметр – массив, и не принимает индексы, и возвращает одиночный элемент массива.

В целом, массивы могут быть многомерными, могут иметь общее содержимое с другими массивами. и могут динамически менять свой

размер после создания (и увеличивать, и уменьшать). Одномерный массив может также иметь *указатель заполнения*.

Многомерные массивы хранят элементы построчно. Это значит, что внутренне многомерный массив хранится как одномерный массив с порядком элементов, соответствующим лексикографическому порядку их индексов. Это важно в двух ситуациях: (1) когда массивы с разными измерениями имеют общее содержимое, и (2) когда осуществляется доступ к очень большому массиву в виртуальной памяти. (Первая ситуация касается семантики; вторая — эффективности)

Массив, что не указывает на другой массив, не имеет указателя заполнения и не имеет динамически расширяемого размера после создания называется *простым* массивом. Пользователи могут декларировать то, что конкретный массив будет простым. Некоторые реализации могут обрабатывать простые массивы более эффективным способом, например, простые массивы могут храниться более компактно, чем непростые.

Когда вызывается `make-array`, если один или более из `:adjustable`, `:fill-pointer` и `:displaced-to` аргументов равен истине, тогда является ли результат простым массивом не определено. Однако если все три аргумента равны лжи, тогда результат гарантированно будет простым массивом.

2.5.1 Векторы

В Common Lisp'е одномерные массивы называется *векторами*, и составляют тип `vector` (который в свою очередь является подтипом `array`). Вектора и списки вместе являются *последовательностями*. Они отличаются тем, что любой элемент одномерного массива может быть получен за константное время, тогда как среднее время доступа к компоненту для списка линейно зависит от длины списка, с другой стороны, добавление нового элемента в начала списка занимает константное время, тогда как эта же операция для массива занимает время линейно зависящее от длины массива.

Обычный вектор (одномерный массив, который может содержать любой тип объектов, но не имеющий дополнительных атрибутов) может быть записан с помощью перечисления элементов разделённых пробелом и окружённых `#(` и `)`. Например:

```
#(a b c)                    ;Вектор из трёх элементов
#()                         ;Пустой вектор
#(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47)
                            ;Вектор содержит простые числа меньшие пятидесяти
```

Следует отметить, что когда функция `read` парсит данный синтаксис, она всегда создаёт *простой* массив.

Обоснование: Многие люди рекомендовали использовать квадратные скобки для задания векторов так: `[a b c]` вместо `#(a b c)`. Данная запись короче, возможно более читаема, и безусловно совпадает с культурными традициями в других областях компьютерных наук и математики. Однако, для достижения предельной полезности от пользовательских макросимволов, что расширяют возможности функции `read`, необходимо было оставить некоторые строковые символы для этих пользовательских целей. Опыт использования MacLisp'a показывает, что пользователи, особенно разработчики языков для использования в исследованиях искусственного интеллекта, часто хотят определять специальные значения для квадратных скобок. Таким образом Common Lisp не использует квадратных и фигурных скобок в своём синтаксисе.

Реализации могут содержать специализированные представления массивов для достижения эффективности в случаях, когда все элементы принадлежат одному определённому типу (например, числовому). Все реализации содержат специальные массивы в случаях, когда все элементы являются строковыми символами (или специализированное подмножество строковых символов). Такие одномерные массивы называются *строки*. Все реализации также должны содержать специализированные битовые массивы, которые принадлежат типу `(array bit)`. Такие одномерные массивы называются *битовые векторы*.

2.5.2 Строки

base-string ≡ (vector base-char)
simple-base-string ≡ (simple-array base-char (*))

Реализация может поддерживать другие типы строк. Все функции Common Lisp'а взаимодействуют со строками одинаково. Однако следует отметить, вставка extended character в base string является ошибкой.

`Строковый (string)` тип является подтипом `векторного (vector)` типа.

Строка может быть записана как последовательность символов, с предшествующим и последующим символом двойной кавычки ". Любой символ " или \ в данной последовательности должен иметь предшествующий символ \.

Например:

```
"Foo"                          ;Строка из трёх символов
""                             ;Пустая строка
"\"APL\\360?\" he cried."      ;Строка из двенадцати символов
"|x| = |-x|"                   ;Строка из десяти символов
```

Необходимо отметить, что символ вертикальной черты | в строке не должен быть экранирован с помощью \. Также как и любая двойная кавычка в имени символа, записанного с использованием вертикальных черт, не нуждается в экранировании. Записи с помощью двойной кавычки и вертикальной черты похожи, но используются для разных целей: двойная кавычка указывает на строку, содержащую строковые символы, тогда как вертикальная черта указывает на символ, имя которого содержит последовательность строковых символов.

Строковые символы обрамленные двойными кавычками, считываются слева направо. Индекс символа больше индекса предыдущего символа на 1. Самый левый символ строки имеет индекс 0, следующий 1, следующий 2, и т.д.

Следует отметить, что функция `prin1` будет выводить на печать в данном синтаксисе любой вектор строковых символов (не только простой), но функция `read` при разборе данного синтаксиса будет всегда создавать простую строку.

2.5.3 Битовые векторы

Битовый вектор может быть записан в виде последовательности битов заключённых в строку с предшествующей `#*`; любой разделитель, например, как пробел завершает синтаксис битового вектора. Например:

```
#*10110     ;Пятибитный битовый вектор; нулевой бит 1
#*          ;Пустой битовый вектор
```

Биты записанные после `#*`, читаются слева направо. Индекс каждого бита больше индекса предыдущего бита на 1. Индекс самого левого бита 0, следующего 1 и т.д.

Функция `prin1` выводит любой битовый вектор (не только простой) в этом синтаксисе, однако функция `read` при разборе этого синтаксиса будет всегда создавать простой битовый вектор.

2.6 Хеш-таблицы

Хеш-таблицы предоставляют эффективный способ для связи любого Lisp объекта (*ключа*) с другим объектом. Они предоставляются как примитивы Common Lisp'a, потому что некоторые реализации могут нуждаться в использовании стратегий управления внутренними хранилищами, что создало бы сложности для пользователя в реализации портируемых хеш-таблиц. Хеш-таблицы описаны в главе 16.

2.7 Таблицы символов Lisp парсера (Readtables)

Таблицы символов Lisp парсера является структурой данных, которая отображает символы в синтаксические типы для парсера Lisp выражений. В частности, эта таблица указывает для каждого строкового символа с синтаксисом *макросимвола*, какой макрос ему соответствует. Это механизм, с помощью которого пользователь может

запрограммировать парсер для выполнения ограниченных, но полезных расширений. Смотрите раздел 22.1.5.

2.8 Пакеты

Пакеты являются коллекциями символов, которые предоставлены в качестве пространства имён. Парсер распознает символы с помощью поиска строки в текущем пакете. Пакеты могут использоваться для скрытия имён внутрь модуля от другого кода. Также предоставляются механизмы для экспортирования символов из заданного пакета в главный «пользовательский» пакет. Смотрите главу 11.

2.9 Имена файлов

Имена файлов являются сущностями, с помощью которых Common Lisp программа может взаимодействовать с внешней файловой системой в приемлемой платформонезависимой форме. Смотрите раздел 23.1.1.

2.10 Потоки

Поток является источником или набором данных, обычно строковых символов или байтов. Почти все функции, что выполняют ввод/вывод, делают это в отношении заданного потока. Функция `open` принимает путь к файлу и возвращает поток подключённый к файлу, указанному в параметре. Существует несколько стандартных потоков, которые используются по умолчанию для различных целей. Смотрите главу 21.

Существуют следующие подтипы типы `stream`: `broadcast-stream`, `concatenated-stream`, `echo-stream`, `synonym-stream`, `string-stream`, `file-stream`, и `two-way-stream` непересекающиеся подтипы типа `stream`. Следует отметить, что поток-синоним всегда принадлежит типу `synonym-stream` вне зависимости от того, какой тип у потока, на который он указывает.

2.11 Состояние для генератора псевдослучайных чисел (Random-States)

Объект типа `random-state` используется для хранении информации о состоянии, используемом генератором псевдослучайных чисел. Для боле подробной информации об объектах `random-state` смотрите главу 12.9.

2.12 Структуры

Структуры являются экземплярами определённых пользователем типов данных, которые имеют ограниченное количество именованных полей (свойств). Они являются аналогами записей в Pascal'е. Структуры декларируются с помощью конструкции `defstruct`. `defstruct` автоматически определяет конструктор и функции доступа к полям для нового типа данных.

Различные структуры могут выводится на печать различными способами. Определение типа структуры может содержать процедуру вывода на печать для объектов данного типа (смотрите опцию `:print-function` для `defstruct`). Записью по-умолчанию для структур является:

#S(*имя-структуры*
имя-слота-1 значение-слота-1
имя-слота-2 значение-слота-2
...)

где `#S` указывает на синтаксис структуры, *имя-структуры* является именем (символом) типа данной структуры, каждый *имя-слота* является именем слота (также символ), и каждое соответствующее *значение-слота* — отображением Lisp объекта в данном слоте.

2.13 Функции

Тип `function` не должен пересекаться с `cons` и `symbol`, и таким образом список, у которого *car* элемент это`lambda` не является, честно говоря, типом `function`, ровно как и любой символ.

Однако стандартные Common Lisp'овые функции, которые принимают функциональные аргументы, будут принимать символ или список, у которого *car* элемент является `lambda` и автоматически преобразовывать их в функции. Эти функции включают в себя `funcall`, `apply` и `mapcar`. Такие функции, однако, не принимают лямбда-выражение в качестве функционального аргумента. Таким образом нельзя записать

```
(mapcar '(lambda (x y) (sqrt (* x y))) p q)
```

но можно что-то вроде

```
(mapcar #'(lambda (x y) (sqrt (* x y))) p q)
```

Это изменение сделало недопустимым представление лексических замыканий, как списка, у которого *car* элемент является некоторым специальным маркером.

Значение специальной формы `function` всегда будет принадлежать типу `function`.

2.14 Нечитаемые объекты данных

Некоторые объекты могут быть выведены на печать в виде, который зависит от реализации. Такие объекты не могут быть полностью реконструированы из распечатанной формы, так как они обычно распечатываются в форме информативной для пользователя, но не подходящей для функции `read`: `#<`*`полезная информация`*`.`

В качестве гипотетического пример, реализация может выводить

```
#<stack-pointer si:rename-within-new-definition-maybe #o311037552>
```

для некоторого специфичного типа данных «внутреннего указатель на стек», у которого выводимое отображение включает имя типа, некоторую информацию о слоте стека, и машинный адрес (в восьмеричной система) данного слота.

Смотрите `print-unreadable-object`, макрос, которые выводит объект используя `#<` синтаксис.

2.15 Пересечение, включение и дизъюнктивность типов

Common Lisp'овая иерархия типов данных запутана и намеренно оставлена несколько открытой, так что разработчики могут экспериментировать с новыми типами данных в качестве расширения языка. В этом разделе чётко оговариваются все определённые связи между типами, в том числе отношения подтипа/супертипа, непересекаемость и исчерпывающее разбиение. Пользователь Common Lisp'а не должен зависеть от любых отношений, явно здесь не указанных. Например, недопустимо предположить, что, поскольку число это не комплексное и не рациональное, то оно должно быть `float`. Реализация может содержать другие виды чисел.

В первую очередь нам необходимо определить термины. Если x супертип y, тогда любой объект типа y принадлежит также типу x, и считается, что y подтип x. Если типы x и y не пересекаются, то ни один объект (ни в одно реализации) не может принадлежать одновременно двум этим типам x и y. Типы с a_1 по a_n являются *исчерпывающим множеством* типа x, если каждый a_j является подтипом x, и любой объект обязательно принадлежит одному из типов a_j. a_1 по a_n являются исчерпывающим разбиением, если они также попарно не пересекаются.

- Тип `t` является супертипом всех остальных типов. Каждый объект принадлежит типу `t`.

- Тип `nil` является подтипом любого типа. Объектов типа `nil` не существует.

- Типы `cons`, `symbol`, `array`, `number`, `character`, `hash-table`, `readtable`, `package`, `pathname`, `stream`, `random-state` и любой другой тип, созданный с помощью `defstruct` or `defclass` являются попарно непересекающимися.

  Тип `function` не пересекается с типами `cons`, `symbol`, `array`, `number`, and `character`.

  Тип `compiled-function` является подтипом `function`. Реализация может также содержать другие потипы `function`.

- Типы `real` and `complex` попарно непересекающиеся подтипы `number`.

Обоснование: Может показаться, что `real` и `complex` должны формировать исчерпывающее множество типа `number`. Но это специально сделано не так, для того чтобы расширения Common Lisp'а могли экспериментировать с числовой системой.

- Типы `rational` и `float` попарно непересекающиеся подтипы `real`.

Обоснование: Может показаться, что `rational` и `float` должны формировать исчерпывающее множество типа `real`. Но это специально сделано не так, для того чтобы расширения Common Lisp'а могли экспериментировать с числовой системой.

- Типы `integer` и `ratio` непересекающиеся подтипы `rational`.

Обоснование: Может показаться, что `integer` и `ratio` должны формировать исчерпывающее множество типа `rational`. Но это специально сделано не так, для того чтобы расширения Common Lisp'а могли экспериментировать с числовой системой.

Types `fixnum` and `bignum` do in fact form an exhaustive partition of the type `integer`; more precisely, they voted to specify that the type `bignum` is by definition equivalent to `(and integer (not fixnum))`. This is consistent with the first edition text in section 2.1.1.

I interpret this to mean that implementators could still experiment with such extensions as adding explicit representations of infinity, but such infinities would necessarily be of type `bignum`.

- Типы `short-float`, `single-float`, `double-float` и `long-float` являются подтипами `float`. Любые два из них могут быть не пересекающимися или идентичными. Если идентичные, тогда любые другие типы между ними в перечисленном порядке должны быть также им идентичны (например, если `single-float` и `long-float` идентичны, то `double-float` должен быть также им идентичен).

- Тип `null` является подтипом `symbol`; только один объект `nil` принадлежит типу `null`.

- Типы `cons` и `null` являются исчерпывающими частями типа `list`.

- Тип `standard-char` является подтипом `base-char`. Типы `base-char` и `extended-char` являются исчерпывающими частями `character`.

- Тип `string` является подтипом `vector`. Множество всех типов (`vector` *c*), включает себя такие типы, как например, когда *c* является подтипом `character`.

- Тип `bit-vector` является подтипом `vector`, для `bit-vector` означает (`vector bit`).

- Типы (`vector t`), `string`, и `bit-vector` являются непересекающимися.

- Тип `vector` является подтипом `array`; для всех типов *x*, тип (`vector` *x*) является тем же, что и тип (`array` *x* (`*`)).

- Тип `simple-array` является подтипом `array`.

- Типы `simple-vector`, `simple-string` и `simple-bit-vector` являются непересекающимися подтипами `simple-array`, для них значит `(simple-array t (*))`, множество все типов `(simple-array c (*))`, в котором, например, *c* является подтипом `character`, и `(simple-array bit (*))`, соответственно.

- Тип `simple-vector` является подтипом `vector` и, конечно, подтипом `(vector t)`.

- Тип `simple-string` является подтипом `string`. (Следует отметить, что `string` является подтипом `vector`, `simple-string` не является подтипом `simple-vector`.)

Обоснование: The hypothetical name `simple-general-vector` would have been more accurate than `simple-vector`, but in this instance euphony and user convenience were deemed more important to the design of Common Lisp than a rigid symmetry.

- Тип `simple-bit-vector` является подтипом `bit-vector`. (Следует отметить, что `bit-vector` является подтипом `vector`, `simple-bit-vector` не является подтипом `simple-vector`.)

- Типы `vector` и `list` является непересекающимися подтипами `sequence`.

- Типы `random-state`, `readtable`, `package`, `pathname`, `stream` и `hash-table` являются попарно непересекающимися.

`random-state`, `readtable`, `package`, `pathname`, `stream`, and `hash-table` попарно не пересекаются с другим типами. Смотрите заметку выше.

- Типы `two-way-stream`, `echo-stream`, `broadcast-stream`, `file-stream`, `synonym-stream`, `string-stream` и `concatenated-stream` являются попарно непересекающимися подтипами `stream`.

- Любые два типа созданные с помощью `defstruct` являются непересекающимися, если только один из них не является супертипом для другого, в котором была указанная опция `:include` с именем этого супертипа.

Глава 3

Область и продолжительность видимости

При описании различных возможностей Common Lisp'а очень важными понятиями являются *области и продолжительности видимости*. Эти понятия возникают, когда к некоторому объекту или конструкции необходимо обратиться из некоторой части кода. *Область видимости* объектов отмечает пространственный или текстовый регион, в котором находящаяся внутри программа может обращаться к этим объектам. *Продолжительность видимости* обозначает временной интервал, в течение которого программа может обращаться к данным объектам.

Вот простой пример такой программы:

```
(defun copy-cell (x) (cons (car x) (cdr x)))
```

Областью видимости параметра с именем `x` является тело формы `defun`. Способа сослаться на этот параметр из какого-либо другого места программы нет. Продолжительностью видимости параметра `x` (для какого-нибудь вызова `copy-cell`) является интервал времени, начиная с вызова функции и заканчивая выходом из неё. (В общем случае продолжительность видимости параметра может продлиться и после завершения функции, но в данном простом случае такого не может быть.)

В Common Lisp сущность, на которую можно сослаться из кода, *создаётся* с помощью специальных языковых конструкций, и область и продолжительность видимости описываются в зависимости от этой

конструкции и времени (выполнения конструкции) в которое эта сущность была создана. Для предмета данного описания, термин «сущность» указывает не только на объекты Common Lisp'а такие как символы и cons-ячейки, но и также на связывания переменных (обычных и специальных), ловушки, и `метки переходов`. Важно отметить различие между сущностью и именем для этой сущности. В определение функции, такой как:

```
(defun foo (x y) (* x (+ y 1)))
```

существует только одно имя, `x`, используемое для ссылки на первый параметр процедуры, когда бы они не была вызвана. *Связывание* — это, в частности, экземпляр параметра. Значение связанное с именем `x` зависит не только от области видимости, в которой данная связь возникла (в данном примере в теле функции `foo` связь возникла в области видимости определения параметров функции), но также, в частности, от механизма связывания. (В данном случае, значение зависит от вызова функции, в течение которого создаётся ссылка). Более сложный пример приводится в конце данной главы.

Вот некоторые виды областей и продолжительностей видимости, которые, в частности, полезны при описании Common Lisp'а:

- *Лексическая область видимости.* В ней связи к установленным сущностям могут использоваться только в той части программы, которая лексически (т.е. текст программы) находится в конструкции устанавливающей данные связи. Обычно эта конструкция будет содержать часть, определённую как *тело (body)*, и область видимости всех сущностей будет установлена только в этом теле.

  Например: имена параметров в функции обычно ограничиваются лексической областью видимости.

- *Неограниченная область видимости.* Ссылка на сущность может производится в любом месте программы.

- *Динамическая продолжительность видимости.* Ссылки на сущности могут производится в любое время на интервале между

установкой сущности и явного упразднения сущности. Как правило, сущность упраздняется, когда выполнение конструкции завершается или как-либо прерывается. Таким образом, сущности с динамической продолжительностью видимости подчиняются механизму типа стек, они распараллеливают выполнение кода, вложенного в их конструкции.

Например: `with-open-file` открывает соединение с файлом и создаёт объект потока для отображения этого соединения. Объект потока имеет неограниченную область видимости, но соединение с открытым файлом имеет динамическую продолжительность видимости: когда выполнение в любом, нормальном или аварийном случае, выйдет за рамки конструкции `with-open-file`, поток будет автоматически закрыт.

Например: связывание «специальной (special)» переменной имеет динамическую продолжительность видимости.

- *Неограниченная продолжительности видимости.* Сущность продолжает существовать пока существует возможность ссылаться на неё. (Реализации разрешается удалить сущность, если она может доказать, что ссылка на неё более невозможна. Стратегии сборщика мусора неявно используют такие доказательства.)

  Например: большинство Common Lisp объектов имеют неограниченную продолжительность видимости.

  Например: связывание лексически замкнутых параметров функции имеет неограниченную продолжительность видимости. (В отличие от Algol'a, где связывание лексически замкнутых параметров процедуры имеют динамическую продолжительность видимости.) Определение функции

```
(defun compose (f g)
  #'(lambda (x)
      (funcall f (funcall g x))))
```

при получении двух параметров, немедленно возвращает функции в качестве результата. Связи параметров для `f` и `g` не теряются,

потому что возвращённая функция, когда будет вызвана, будет продолжать ссылаться на эти связи. Таким образом

(funcall (compose #'sqrt #'abs) -9.0)

вернёт значение `3.0`. (Аналогичная процедура не захочет корректно работать в типичной реализации Algol'a или, даже, в большинстве диалектов Lisp'a.)

В дополнение к вышеназванным терминам, удобно определить *динамическую область видимости*, которая означает *неограниченную область видимости и динамическую продолжительность видимости*. Следовательно мы говорим о «специальных (special)» переменных, как об имеющих динамическую область видимости или будучи динамически замкнутых FIXME, потому что они имеют неограниченную область видимости и динамическую продолжительность видимости: к специальным переменным можно сослаться из любой точки программы на протяжении существования их связываний.

Термин «динамическая область видимости» некорректен. Как бы то ни было это и устоялось, и удобно.

Сказанное выше не рассматривает возможность *скрытия (shadowing)*. Далёкие (FIXME) ссылки на сущности осуществляются с использованием *имён* того или иного типа. Если две сущности имеют одинаковое имя, тогда второе имя может скрыть первое, в таком случае ссылка с помощью этого имени будет осуществлена на вторую сущность и не может быть осуществлена на первую.

В случае лексической области видимости, если две конструкции, которые устанавливают сущности с одинаковыми именами, расположены в тексте одна внутри другой, тогда ссылки внутри внутренней конструкции указывают на сущности внутренней конструкции, то есть внутренние сущности скрывают внешние. Вне внутренней конструкции, но внутри внешней конструкции ссылки указывают на сущности, установленные внешней конструкцией.

```
(defun test (x z)
  (let ((z (* x 2))) (print z))
  z)
```

Связывание переменной `z` с помощью конструкции `let` скрывает связывание одноимённого параметра функции `test`. Ссылка на переменную `z` в форме `print` указывает на `let` связывание. Ссылка на `z` в конце функции указывает на параметр с именем `z`.

В случае динамической продолжительности видимости, если временные интервалы двух сущностей перекрываются, тогда они будут обязательно вложенными один в другого. Это свойство Common Lisp дизайна.

Заметка для реализации: Behind the assertion that dynamic extents nest properly is the assumption that there is only a single program or process. Common Lisp does not address the problems of multiprogramming (timesharing) or multiprocessing (more than one active processor) within a single Lisp environment. The documentation for implementations that extend Common Lisp for multiprogramming or multiprocessing should be very clear on what modifications are induced by such extensions to the rules of extent and scope. Implementors should note that Common Lisp has been carefully designed to allow special variables to be implemented using either the "deep binding" technique or the "shallow binding" technique, but the two techniques have different semantic and performance implications for multiprogramming and multiprocessing.

Ссылка по имени на сущность с динамической продолжительностью жизни всегда указывает на сущность с этим именем, что была установлена наипозднейшей и ещё не была упразднена. Например:

```
(defun fun1 (x)
  (catch 'trap (+ 3 (fun2 x))))

(defun fun2 (y)
  (catch 'trap (* 5 (fun3 y))))

(defun fun3 (z)
  (throw 'trap z))
```

Рассмотрим вызов `(fun1 7)`. Результатом будет `10`. Во время выполнения `throw`, существует две ловушки с именем `trap`: одна установлена в процедуре `fun1`, и другая — в `fun2`. Более поздняя в `fun2`, и тогда, из формы `catch`, что в `fun2`, возвращается значение `7`.

Рассматриваемая из `fun3`, `catch` в `fun2` скрывает одноимённую в `fun1`. Если бы `fun2` была определена как

```
(defun fun2 (y)
  (catch 'snare (* 5 (fun3 y))))
```

тогда бы две ловушки имели разные имена, и в таком случае одна из них из `fun1` не была бы скрыта. Результатом бы стало `7`.

Как правило, данная книга по простому рассказывает об областях видимости и продолжительности сущности, возможность скрытия оставляется без рассмотрения.

Далее важные правила области и продолжительности видимости в Common Lisp'e:

- Связывания переменных обычно имеют лексическую область видимости и неограниченную продолжительность видимости.

- Связывания переменных, для которых декларировано `dynamic-extent` также имеют лексическую область видимости и неограниченную продолжительность, но объекты, которые являются значениями этих связываний могут иметь динамическую продолжительность видимости.

- Связывания имён переменных с символом макроса с помощью `symbol-macrolet` имеют лексическую область видимости и неограниченную продолжительность видимости.

- Связывания переменных, которые задекларированы быть `специальными (special)`, имеют динамическую область видимости (неограниченную область видимости и динамическую продолжительность).

- Связывания имён функций устанавливаются, например, формами `flet` и `labels` и имеют лексическую область видимости и неограниченную продолжительность.

- Связывания имён функций, для которых задекларировано `dynamic-extent`, также имеют лексическую область видимости и неограниченную продолжительность, но объекты функции, которые являются значениями для данных связываний могут иметь динамическую продолжительность видимости.

- Связывания имён функций с макросами, установленными с помощью `macrolet` имеют лексическую область видимости и неограниченную продолжительность.

- Обработчики условий и перезапусков (condition and restarts) имеют динамическую область видимости (смотрите главу 28).

- Ловушка установленная с помощью специальных форм `catch` или `unwind-protect` имеет динамическую область видимости.

- Точка выхода установленная с помощью конструкции `block` имеет лексическую область видимости и динамическую продолжительность. (Такие точки выхода, также устанавливаются с помощью `do`, `prog` и другими конструкциями для итераций.)

- Цели для `go`, устанавливающиеся с помощью `tagbody` и именующиеся с помощью тегов в `tagbody`, на которые указывает `go`, имеют лексическую область видимости и динамическую продолжительность. (Такие `go` цели могут также появляться как теги в телах `do`, `prog` и других конструкций для итераций.)

- Именованные константы, такие как `nil` и `pi` имеют неограниченную область видимости и неограниченную продолжительность.

Правила для лексической области видимости подразумевают, что лямбда-выражения (анонимные функции), появляющиеся в `function`, будут являться «замыканиями» над этими неспециальными (non-special) переменными, которые видимы для лямбда-выражения. Это значит, что функция предоставленная лямбда-выражением может ссылаться на любую лексически доступную неспециальную (non-special) переменную и получать корректное значение, даже если выполнение уже вышло

из конструкции, которая устанавливала связи. Пример `compose`, рассмотренный в данной главе ранее, предоставлял изображение такого механизма. Правила также подразумевают, что связывания специальных переменных не «замыкаются», как может быть в некоторых других диалектах Lisp'а.

Конструкции, которые используют лексическую область видимости генерируют новое имя для каждой устанавливаемой сущности при каждом исполнении. Таким образом, динамическое скрытие не может произойти (тогда как лексическое может). Это, в частности, важно, когда используется динамическая продолжительность видимости. Например:

```
(defun contorted-example (f g x)
  (if (= x 0)
      (funcall f)
      (block here
        (+ 5 (contorted-example g
                                #'(lambda ()
                                    (return-from here 4))
                                (- x 1))))))
```

Рассмотрим вызов `(contorted-example nil nil 2)`. Он вернёт результат `4`. Во время исполнения, `contorted-example` будет вызвана три раза, чередуясь с двумя блоками:

```
(contorted-example nil nil 2)

  (block here1 ...)

    (contorted-example nil #'(lambda () (return-from here1 4)) 1)

      (block here2 ...)

        (contorted-example #'(lambda () (return-from here1 4))
                           #'(lambda () (return-from here2 4))
                           0)
```

(funcall f)
    where f ⇒ #'(lambda () (return-from here$_1$ 4))

(return-from here$_1$ 4)

В время выполнения `funcall` существует две невыполненные точки выхода `block`, каждая с именем `here`. В стеке вызовов выше, эти две точки для наглядности проиндексированы. Форма `return-from`, выполненная как результат операции `funcall`, ссылается на *внешнюю* невыполненную точку выхода (`here`$_1$), но не на (`here`$_2$). Это следствие правил лексических областей видимости: форма ссылается на ту точку выхода, что видима по тексту в точке вызова создания функции (здесь отмеченной с помощью синтаксиса `#'`). (FIXME)

Если в данном примере, изменить форму `(funcall f)` на `(funcall g)`, тогда значение вызова `(contorted-example nil nil 2)` будет `9`. Значение измениться по сравнению с предыдущим разом, потому что `funcall` вызовет выполнение `(return-from here`$_2$ `4)`, и это в свою очередь вызовет выход из внутренней точки выхода (`here`$_2$). Когда это случиться, значение `4` будет возвращено из середины вызова `contorted-example`, к нему добавится `5` и результат окажется `9`, и это значение вернётся из внешнего блока и вообще из вызова `contorted-example`. Цель данного примера, показать что выбор точки выхода зависит от лексической области, которая была захвачена лямбда-выражением, когда вызывался код создания этой анонимной функции.

Эта функция `contorted-example` работает только потому, что функция с именем `f` вызывается в процессе продолжительности действия точки выхода. Точки выхода из блока ведут себя, как связывания неспециальных (non-special) переменных в имеющимся лексическом окружении, но отличаются тем, что имеют динамическую продолжительность видимости, а не неограниченную. Как только выполнение покинет блок с этой точкой выхода, она перестанет существовать. Например:

```
(defun illegal-example ()
  (let ((y (block here #'(lambda (z) (return-from here z)))))
    (if (numberp y) y (funcall y 5))))
```

Можно предположить, что вызов `(illegal-example)` вернёт `5`: Форма `let` связывает переменную `y` со значением выполнения конструкции `block`; её значение получится равным анонимной функции. Так как `y` не является числом, она вызывается с параметром `5`. `return-from` тогда должны вернуть данное значение с помощью точки выхода `here`, тогда осуществляется выход из блока *ещё раз* и `y` получает значение `5`, которое будучи числом, возвращается в качестве значения для `illegal-example`.

Рассуждения выше неверны, потому что точки выхода определяемые в Common Lisp'е имеют динамическую продолжительность видимости. Аргументация верна только до вызова `return-from`. Вызов формы `return-from` является ошибкой, *не потому что* она не может сослаться на точку выхода, а потому что она корректно ссылается на точку выхода *и* эта точка выхода уже была упразднена.

Глава 4

Спецификаторы типов

В Common Lisp'е типы указываются с помощью символов и списков. Они называются *спецификаторами типов*. Символы задают предопределённые классы объектов, тогда как списки обычно указывают на комбинации или уточнения простых типов. Символы и списки могут быть также аббревиатурами для других типов.

4.1 Символы как спецификаторы типов

Символы для типов жестко заданы системой, включая те, что перечислены в таблице 4.1. Когда определяется структура с использованием `defstruct`, имя структуры также автоматически становиться типом.

4.2 Списки как спецификаторы типов

Если типа задаётся списком, *car* элемент данного списка является символом, и остаток списка — вспомогательной информацией. В большинстве случаев вспомогательная информация может быть *неопределена*. Неопределённая дополнительная информация указывается с помощью `*`. Например, для полного описания векторного типа, должны быть указаны тип элементов и их количество:

(vector double-float 100)

Таблица 4.1: Стандартные символы для обозначения типов

array	fixnum	package	simple-string
atom	float	pathname	simple-vector
bignum	function	random-state	single-float
bit	hash-table	ratio	standard-char
bit-vector	integer	rational	stream
character	keyword	readtable	string
	list	sequence	
compiled-function	long-float	short-float	symbol
complex	nil	signed-byte	t
cons	null	simple-array	unsigned-byte
double-float	number	simple-bit-vector	vector

Для указания неопределённой длины, можно записать:

(vector double-float *)

Для указания неопределённого типа элемента, можно записать:

(vector * 100)

Можно также оставить неопределёнными и тип элемента, и длину:

(vector * *)

Допустим, что два спецификатора являются одинаковыми за исключением того, что первый содержит `*`, а второй содержит конкретное значение. Тогда второй тип является подтипом первого типа.

Для удобства существует следующее правило: если список в конце содержит незаданные элементы (`*`), то они могут быть опущены. Если все, за исключением первого, элементы не специфицированы, то данный список может быть упрощён вплоть до упразднения скобок и превращения его в простой символ, который был в *car*. Например, `(vector * *)` может быть записан как `(vector)`, а затем как `vector`.

4.3 Предикаты как спецификаторы типов

Список `(satisfies` *`predicate-name`*`)` задаёт тип, которому принадлежит множество объектов, удовлетворяющие предикату *predicate-name*. *predicate-name* должен быть символом, указывающим на глобальную функцию с одним аргументом. (Требуется именно имя, так как лямбда-выражение недопустимо в связи с проблемами видимости.) Например, тип `(satisfies numberp)` является тем же, что и `number`. Вызов `(typep x '(satisfies p))` применяет `p` к `x` и возвращает `t`, если результат true, и `nil` в противном случае.

Порождать побочные эффекты не является хорошей идеей для предиката, что указан в `satisfies`.

4.4 Комбинированные спецификаторы типов

Следующие списки-спецификаторы типов, определяют новый тип в терминах других типов или объектов.

`(member` *`object1 object2 ...`*`)` Такая запись обозначает тип, как множество содержащее определённый набор объектов. Объект принадлежит данному типу тогда и только тогда, когда он равен `eql` одному из заданных объектов. FIXME.

`(eql` *`object`*`)` Такой специализатор может быть использован для определения CLOS методов. Смотрите раздел 27.1.6 и `find-method`. Он задаёт множество из одного объекта. Объект принадлежит такому типу тогда и только тогда, когда он `eql` для первого *объекта*. Несмотря на то, что `(eql` *`объект`*`)` обозначает то же, что и `(member` *`объект`*`)`, только `(eql` *`объект`*`)` может быть использован для определения CLOS метода.

`(not` *`type`*`)` Такая запись задаёт тип множества объектов, которые *не* являются типом *type*.

`(and` *`type1 type2 ...`*`)` Такая запись задаёт пересечение указанных типов.

Когда пердикат `typep` обрабатывает спецификатор типа `and`, он производит проверку на принадлежность к каждому подтипу слева направо и моментально останавливает проверку, в случае первого случившегося отрицательного результата. Таким образом, спецификатор `and` ведает себя подобно форме `and`. Цель такого сходства — позволить спецификатору типа `satisfies` зависеть от фильтрации предыдущим спецификатором типа. Например, предположим, что есть функция `primep`, которая принимает целое и возвращает `t`, если оно простое. Также предположим, что является ошибочным передавать любой объект, не являющийся целым, в `primep`. Тогда спецификатор будет выглядеть так:

```
(and integer (satisfies primep))
```

никогда не вызовет ошибку, так как функция `primep` никогда не будет вызвана с объектом, который не удовлетворяет предыдущему типу integer.

`(or` *`type1 type2 ...`*`)` Такая запись обозначает объединение типов. Например, тип `list` совпадает с `(or null cons)`. Также, значение, возвращаемое функцией `position` всегда принадлежит типу `(or null (integer 0 *))` (или `nil` или неотрицательное число).

Также как и для `and`, когда `typep` обрабатывает спецификатор типа `or`, он поочерёдно проверяет каждый подтип объединения слева направо и завершает обработку, как только принадлежность подтипу установлена.

4.5 Уточняющие спецификаторы типов

Некоторые списки, представляющие типы, с помощью символов могут быть более специализированы. Такие подробности могут

быть отражены, как более эффективная реализация. Например, предположим что `(array short-float)`. Реализация A, может выбрать специализированное представление для массива коротких с плавающей точкой, а реализация B может выбрать более общее представление.

Если вы хотите создать массив в целях хранения только коротких с плавающей точкой, вы можете опционально указать для `make-array` тип элементов `short-float`. Это *не потребует* от `make-array` создать объект типа `(array short-float)`, но это просто позволит ей выбрать родственный тип. Запрос можно объяснить так: «Предоставь наиболее специализированный массив, который может хранить короткие с плавающей точкой, который только может предоставить реализация». Реализация A тогда предоставит специализированный массив типа `(array short-float)`, а реализация B — простой массив типа `(array t)`.

На вопрос, действительно ли тип созданного массива `(array short-float)`, реализация A ответит «да», но реализация B ответит «нет». Это свойство `make-array` и подобных ей функций: то, что вы просите, необязательно является тем, что вы получите.

X3J13 voted in January 1989 to eliminate the differing treatment of types when used “for discrimination” rather than “for declaration” on the grounds that implementors have not treated the distinction consistently and (which is more important) users have found the distinction confusing.

As a consequence of this change, the behavior of `typep` and `subtypep` on `array` and `complex` type specifiers must be modified. See the descriptions of those functions. In particular, under their new behavior, implementation B would say “yes,” agreeing with implementation A, in the discussion above.

Note that the distinction between declaration and discrimination remains useful, if only so that we may remark that the specialized (list) form of the `function` type specifier may still be used only for declaration and not for discrimination.

X3J13 voted in June 1988 to clarify that while the specialized form of the `function` type specifier (a list of the symbol `function` possibly followed by argument and value type specifiers) may be used only for declaration, the symbol form (simply the name `function`) may be used for discrimination.

Далее перечислены возможные имена типов, которые задаются списком:

`(array` *`element-type size`*`)` Такая запись обозначает множество специализированных массивов, элементы которых принадлежат типу *element-type* и размер которых равен *size*. *element-type* должен быть корректным спецификатором типа или не уточнён с помощью `*`. *size* может быть неотрицательным целым, определяющим размер массива, может быть списком неотрицательных целых, определяющих размер каждого измерения (размер какого-либо измерения может быть не указан, `*`) или может быть не указан `*`. Например,

```
(array integer 3)          ;Трёхэлементный массив целых
(array integer (* * *))     ;Трёхмерный массив целых
(array * (4 5 6))          ;Трёхмерный массив, размеры измерений 4,5,6
(array character (3 *))     ;Двумерный массив символов
                          ; у которого только три строки
(array short-float ())      ;Ранг массива равен нулю, массив содержит
                          ; короткие с плавающие точкой
```

Следует отметить, что `(array t)` является правильным подмножеством `(array *)`. Причиной тому является то. что `(array t)` это множество массивов, которые могут содержать любой Common Lisp объект (элементы типа `t`, которые включают все элементы). С другой стороны, `(array *)` является множеством всех массивов, включая, например, массивы, которые могут хранить только строковые символы. Сейчас `(array character)` не является подмножеством `(array t)`; два множества фактически непересекаются, потому что `(array character)` не является множеством всех массивов, которые могут хранить строковые символы, а является множеством массивов, которые специализированы хранить именно символы и никакие другие объекты FIXME. Поэтому проверка, может ли массив `foo` хранить строковые символы, не может быть такой:

```
(typep foo '(array character))
```

, а должна быть такой:

(subtypep 'character (array-element-type foo))

Смотрите `array-element-type`. X3J13 voted in January 1989 to change `typep` and `subtypep` so that the specialized `array` type specifier means the same thing for discrimination as for declaration: it encompasses those arrays that can result by specifying *element-type* as the element type to the function `make-array`. Under this interpretation `(array character)` might be the same type as `(array t)` (although it also might not be the same). See `upgraded-array-element-type`. However,

(typep foo '(array character))

is still not a legitimate test of whether the array `foo` can hold a character; one must still say

(subtypep 'character (array-element-type foo))

to determine that question.

X3J13 also voted in January 1989 to specify that within the lexical scope of an array type declaration, it is an error for an array element, when referenced, not to be of the exact declared element type. A compiler may, for example, treat every reference to an element of a declared array as if the reference were surrounded by a `the` form mentioning the declared array element type (*not* the upgraded array element type). Thus

```
(defun snarf-hex-digits (the-array)
  (declare (type (array (unsigned-byte 4) 1) the-array))
  (do ((j (- (length array) 1) (- j 1))
       (val 0 (logior (ash val 4)
                      (aref the-array j))))
      ((< j 0) val)))
```

may be treated as

```
(defun snarf-hex-digits (the-array)
  (declare (type (array (unsigned-byte 4) 1) the-array))
  (do ((j (- (length array) 1) (- j 1))
       (val 0 (logior (ash val 4)
                      (the (unsigned-byte 4)
                           (aref the-array j)))))
      ((< j 0) val)))
```

The declaration amounts to a promise by the user that the `aref` will never produce a value outside the interval 0 to 15, even if in that particular implementation the array element type `(unsigned-byte 4)` is upgraded to, say, `(unsigned-byte 8)`. If such upgrading does occur, then values outside that range may in fact be stored in `the-array`, as long as the code in `snarf-hex-digits` never sees them.

As a general rule, a compiler would be justified in transforming

(aref (the (array *elt-type* ...) *a*) ...)

into

(the *elt-type* (aref (the (array *elt-type* ...) *a*) ...)

It may also make inferences involving more complex functions, such as `position` or `find`. For example, `find` applied to an array always returns either `nil` or an object whose type is the element type of the array.

X3J13 voted in January 1989 to change `typep` and `subtypep` so that the specialized `array` type specifier means the same thing for discrimination as for declaration: it encompasses those arrays that can result by specifying *element-type* as the element type to the function `make-array`. Under this interpretation `(array character)` might be the same type as `(array t)` (although it also might not be the same). See `upgraded-array-element-type`. However,

(typep foo '(array character))

is still not a legitimate test of whether the array `foo` can hold a character; one must still say

(subtypep 'character (array-element-type foo))

to determine that question.

As a general rule, a compiler would be justified in transforming

(aref (the (array *elt-type* ...) *a*) ...)

into

(the *elt-type* (aref (the (array *elt-type* ...) *a*) ...)

It may also make inferences involving more complex functions, such as `position` or `find`. For example, `find` applied to an array always returns either `nil` or an object whose type is the element type of the array.

`(simple-array` *element-type sizes*`)` Данная запись эквивалентна `(array` *element-type sizes*`)` за исключением того, что дополнительно определяет, что объекты будут *простыми* массивами (смотрите раздел 2.5).

`(vector` *element-type size*`)` Такой тип обозначает множество специализированный одномерных массивов, все элементы которых принадлежат типу `element-type` и size которого равен `size`. Такой тип полностью эквивалентен `(array` *element-type* `(`*size*`))`. Например:

```
(vector double-float)     ;Векторы двойных
                          ; чисел с плавающей точкой
(vector * 5)              ;Векторы длинной пять элементов
(vector t 5)              ;Общие векторы длинной пять элементов FIXME
(vector (mod 32) *)       ;Вектора целых чисел между 0 и 31
```

Тип `string` является объединение одно или более специализированных типов векторов, а именно всех векторов, тип элементов которых является подтипом `character`.

`(simple-vector` *`size`*`)` Такая запись означает то же, что и `(vector t` *`size`*`)`.

`(complex` *`type`*`)` Каждый элемент такого типа является комплексным числом, у которого действительная и мнимая части принадлежат типу *type*. For declaration purposes, this type encompasses those complex numbers that can result by giving numbers of the specified type to the function `complex`; this may be different from what the type means for discrimination purposes. As an example, Gaussian integers might be described as `(complex integer)`, even in implementations where giving two integers to the function `complex` results in an object of type `(complex rational)`.

X3J13 voted in January 1989 to change `typep` and `subtypep` so that the specialized `complex` type specifier means the same thing for discrimination purposes as for declaration purposes. See `upgraded-complex-part-type`.

`(function (`*`argument-type-1 argument-type-2 ...`*`)` *`value-type`*`)` Этот тип может использоваться только для декларации и не может для распознавания; `typep` будет сигнализировать ошибку, если ей будет передан такой спецификатор типа. Каждый элемент такого типа является функцией, которая принимает аргументы типов перечисленных с помощью форм *argument-type-j* и возвращает значение типа *value-type*. В форме типов аргументов могут использоваться маркеры `&optional`, `&rest` и `&key`. *value-type* может быть спецификатором типа `values` в случае, если функция возвращает несколько значений.

`(values` *`value-type-1 value-type-2 ...`*`)` Данный тип используется только в двух случаях: только как *value-type* для спецификатора типа *функции* и в специальной форме `the`. Данный спецификатор используется для задания типов в случаях возврата нескольких значений. В списке с типами могут использоваться маркеры `&optional`, `&rest` и `&key`.

4.6 Аббревиатуры для спецификаторов типов

Следующие спецификаторы типов по большей части являются аббревиатурами для других типов, которые долго печатать, например, в функции `member`.

`(integer` *`low high`*`)` Задаёт целые числа между значениями *low* и *high*. *Low* и *high* должны быть каждое или целое, или список с одним целым, или не заданы. Целое задаёт включаемое граничное значение, список из целого задаёт невключаемое граничное значение и `*` означает отсутствие граничного значения, и тем самым задаёт границы минус или плюс бесконечность соответственно. Тип `fixnum` является именем для `(integer` *`low high`*`)`, в котором *high* и *low* значения зависят от реализации (см. `most-negative-fixnum` и `most-positive-fixnum`). Тип `(integer 0 1)` оказался так полезен, что имеет отдельное имя `bit`.

`(mod` *`n`*`)` Задаёт множество неотрицательных целых меньших чем `n`. Является эквивалентом для `(integer 0` $n-1$`)` или `(integer 0 (`*`n`*`))`.

`(signed-byte` *`s`*`)` Задаёт множество целых, которые могут быть представлены в виде байта с `s` количеством бит. Является эквивалентом для `(integer` -2^{s-1} $2^{s-1}-1$`)`. `signed-byte` или `(signed-byte *)` являются тем же, что и `integer`.

`(unsigned-byte` *`s`*`)` Задаёт множество неотрицательных целых, которые могут быть представлены в виде байта с `s` количеством бит. Является эквивалентом для `(mod` 2^s`)`, и для `(integer 0` 2^s-1`)`. `unsigned-byte` или `(unsigned-byte *)` являются тем же, что и `(integer 0 *)`, а именно, множеством неотрицательных целых.

`(rational` *`low high`*`)` Задаёт рациональные числа между значениями *low* и *high*. Наименьшее и наибольшее могут быть рациональными, списком из одного рационального, или не заданы. Рациональное число задаёт включаемое граничное значение, список из рационального задаёт невключаемое граничное значение и `*`

означает, что предела для множества нет, и значением может быть минус или плюс бесконечность соответственно.

`(float` *`low high`*`)` Задаёт числа с плавающей точкой между *low* и *high*. Наименьшее и наибольшее могут быть числами с плавающей точкой, списком из одного такого числа, или не заданы. Рациональное число задаёт включаемое граничное значение, список из рационального задаёт невключаемое граничное значение и `*` означает, что предела для множества нет, и значением может быть минус или плюс бесконечность соответственно.

Таким же образом определяются следующие типы:

(short-float *low high*)
(single-float *low high*)
(double-float *low high*)
(long-float *low high*)

В этих случаях границы должны быть в том же формате, что и сам тип.

`(real` *`low high`*`)` Задаёт действительные числа между значениями *low* и *high*. Пределы *low* и *high* должны каждый быть действительными числами, или списками одного действительного числа, или не заданы. Действительное число задаёт включаемое граничное значение, список из действительного числа задаёт невключаемое граничное значение и `*` означает, что предела для множества нет, и значением может быть минус или плюс бесконечность, соответственно.

`(base-string` *`size`*`)` Обозначает то же, что и `(vector base-char` *`size`*`)`: множество базовых строк определённого размера.

`(simple-base-string` *`size`*`)` Обозначает то же, что и `(simple-array base-char` *`size`*`)`: множество простых базовых строк определённого размера.

`(bit-vector` *`size`*`)` Обозначает то же, что и `(array bit (`*`size`*`))`: множество битовых векторов определённого размера.

`(simple-bit-vector` *`size`*`)` Обозначает то же, что и `(simple-array bit (`*`size`*`))`: множество битовых векторов определённого размера.

4.7 Определение новых спецификаторов

Новые спецификаторы создаются двумя способами. Первый, определение нового типа структуры с помощью `defstruct` автоматически создаёт новый тип с именем, как у структуры. Второй, использование `deftype` для создания новых аббревиатур для спецификаторов типов.

[Макрос] **deftype** name lambda-list
[[{declaration}* | doc-string]] {form}*

Данный макрос весьма схож с формой `defmacro`: *name* является символом, который будет определять имя будущего спецификатора типа, `lambda-list` является лямбда списком (и может содержать маркеры `&optional` и `&rest`) и *forms* составляют тело функции. Если мы рассмотрим спецификатор типа, как список, содержащий имя и несколько форм аргументов, то формы аргументов (невычисленные) будут связаны с параметрами из лямбда списка *lambda-list*. Затем будут выполнены формы тела, как неявный `progn`, и значение последней формы будет интерпретировано, как новый спецификатор типа, для которого исходный спецификатор является аббревиатурой. *name* возвращается, как значение формы `deftype`.

`deftype` отличается от `defmacro` в том, что если для необязательного `&optional` параметра значение по умолчанию *initform* не задано, то используется `*`, а не `nil`.

Если указана необязательная строка документации *doc-string*, тогда она присоединяется к имени `name`, как строка документации типа `type`. Смотрите `documentation`.

Вот несколько примеров использования `deftype`:

```
(deftype mod (n) '(integer 0 (,n)))

(deftype list () '(or null cons))
```

```
(deftype square-matrix (&optional type size)
  "SQUARE-MATRIX includes all square two-dimensional arrays."
  `(array ,type (,size ,size)))
```

(square-matrix short-float 7) означает (array short-float (7 7))

(square-matrix bit) означает (array bit (* *))

Если имя типа заданного с помощью `deftype` используется просто как символ спецификатора типа, тогда это имя интерпретируется, как список, задающий тип с аргументами по-умолчанию `*`. Например, используя код выше, `square-matrix` будет означать `(array * (* *))`, то есть множество двумерных

В таком случае к несчастью нарушается правило о том, что количество строк в матрице должно совпадать с количеством столбцов. `(square-matrix bit)` имеет такую же проблему. Лучшим решением будет:

```
(defun equidimensional (a)
  (or (< (array-rank a) 2)
      (apply #'= (array-dimensions a))))

(deftype square-matrix (&optional type size)
  `(and (array ,type (,size ,size))
        (satisfies equidimensional)))
```

Тело функции типа, определяемой с помощью `deftype`, неявно оборачивается в конструкцию `block`, имя которой совпадает с именем определеяемого типа. Таким образом, для выхода из функции может использоваться `return-from`.

Тогда как обычно эта форма используется на верхнем уровне, ее можно использовать и внутри других форм. Например, `deftype` может определять функцию типа внутри лексического, а не глобального окружения.

4.8 Приведение типов

Следующие функции могут быть использованы для преобразования объекта в эквивалентный объект другого типа.

[Функция] **coerce** *object result-type*

result-type должен быть спецификатором типа. *object* будет конвертирован в «эквивалентный» объект заданного типа. Если преобразование не может быть осуществлено, будет сигнализирована ошибка. В частности, `(coerce x 'nil)` всегда сигнализирует ошибку. Если *object* уже принадлежит заданному типу, это проверяется предикатом `typep`, тогда данный объект будет возвращён. В целом, невозможно преобразовать любой объект в объект любого другого типа. Допускаются только следующие преобразования.

- Любой тип последовательности может быть преобразован в другой тип последовательности. Новый тип последовательности будет содержать все объекты из старой последовательности (если это невозможно, тогда возникнет ошибка). Если `result-type` задан как `array`, тогда будет использоваться `(array t)`. Также может использоваться специализированный тип такой, как `string` или `(vector (complex short-float))`; конечно, результат в зависимости от реализации может быть более общим типом. Элементы новой последовательности будут равны `eql` соответствующим элементам старой последовательности. Если последовательность уже принадлежит заданному типу, она может быть просто возвращена без копирования. В таком случае `(coerce` *`последовательность тип`*`)` отличается от `(concatenate` *`тип последовательность`*`)`, так как последняя требует копирования аргумента *последовательность*.

  (coerce '(a b c) 'vector) ⇒ #(a b c)

  `coerce` должен сигнализировать ошибку, если новая тип новой последовательности определеяет некоторое количество элементов, а длина старой последовательности от него отличается.

Если *result-type* является строкой (`string`), тогда под ним понимается `(vector character)`, и под `simple-string` понимается `(simple-array character (*))`.

- Любое некомплексное число может быть приведено к `short-float`, `single-float`, `double-float` или `long-float`. Если тип указан с плавающей точкой и *объект* не является числом с плавающей точкой, тогда объект преобразовывается в `single-float`.

  (coerce 0 'short-float) ⇒ 0.0S0
  (coerce 3.5L0 'float) ⇒ 3.5L0
  (coerce 7/2 'float) ⇒ 3.5

- Любое число может быть приведено к комплексному. Если число ещё не является комплексным, тогда мнимая часть будет равна нулю, который будет преобразован в тип, соответствующий типу действительной части. (Если полученная действительная часть является рациональным числом, тогда результат немедленно будет преобразован из комплексного обратно в рациональный.)

  (coerce 4.5s0 'complex) ⇒ #C(4.5S0 0.0S0)
  (coerce 7/2 'complex) ⇒ 7/2
  (coerce #C(7/2 0) '(complex double-float))
  ⇒ #C(3.5D0 0.0D0)

- Любой объект может быть приведён к типу `t`.

  (coerce x 't) ≡ (identity x) ≡ x

- Символ или лямбда-выражение может быть преобразовано к функции. Символ приводится к типу `function`, как если бы к нему была применена функция `symbol-function`. Если символ не связан (`fboundp` symbol -> false), или символ связан с макросом или специальной формой сигнализируется ошибка. Список *x*, чей *car* является символом `lambda` приводится к функции, как если бы было вычислено выражение `(eval '#'`*x*`)`, или `(eval (list 'function` *x*`))`.

Приведение чисел с плавающей точкой к рациональным и рациональных целым *не* предоставляется в связи с проблемами округления. Для этого могут использоваться функции `rational`, `rationalize`, `floor`, `ceiling`, `truncate` и `round`. Также не предоставляется приведение строковых символов к целым числам. В этих целях можно использовать `char-code` или `char-int`.

4.9 Определение типа объекта

Следующие функции могут быть использованы для получения спецификатора, обозначающего тип заданного объекта.

[Функция] **type-of** *object*

Функция имеет следующие ограничения.

- Пусть *x* является объектом, и `(typep` *x type*`)` вычисляется в истину, и *тип* один из:

array	float	package	sequence
bit-vector	function	pathname	short-float
character	hash-table	random-state	single-float
complex	integer	ratio	stream
condition	long-float	rational	string
cons	null	readtable	symbol
double-float	number	restart	vector

- Для любого объекта *x*, `(subtypep (type-of x) (class-of x))` должно вернуть значения `t` и `t`.

- Для каждого объекта *x*, `(typep x (type-of x))` должно быть true. (Это означает, что `type-of` никогда не может вернуть `nil`, так как нет объектов принадлежащий типу `nil`.)

- `type-of` никогда не возвращает `t` и никогда не использует спецификаторы типа `satisfies`, `and`, `or`, `not` или `values` в качестве результата.

- Для объектов CLOS метакласса `structure-class` или `standard-class`, `type-of` возвращает имя класса, получаемое с помощью `class-of`, если оно имеется, в противном случае возвращается сам класс. В частности, для любого объекта созданного с помощью `defstruct` функции-конструктора, и `defstruct` имело имя *name* и не имело опции `:type`, `type-of` вернёт *name*.

В качестве примера, `(type-of "acetylcholinesterase")` может вернуть `string` или `simple-string` или `(simple-string 20)`, но не `array` или `simple-vector`. Другой пример, `(type-of 1729)` может вернуть `integer` или `fixnum` или `(signed-byte 16)` или `(integer 1729 1729)` или `(integer 1685 1750)` или даже `(mod 1730)`, но не `rational` или `number`, потому что

```
(typep (+ (expt 9 3) (expt 10 3)) 'integer)
```

является истиной, `integer` содержится в списке упомянутом выше, и

```
(subtypep (type-of (+ (expt 1 3) (expt 12 3))) 'integer)
```

будет ложью, если `type-of` вернёт `rational` или `number`.

4.10 Подбираемый тип

Common Lisp содержит функции, с помощью которых программы смогут установить, как данная реализация будет *подбирать* тип, когда создаёт массив для некого заданного типа элементов, или комплексное число с заданными типами частей.

[Функция] **upgraded-array-element-type** *type*

Функция возвращает спецификатор типа, наиболее близкий к указанному, как если бы последний использовался в функции `make-array`. Результат обязательно является супертипом для заданного *type*. Кроме того, если тип *A* является подтипом *B*, тогда `(upgraded-array-element-type A)` является подтипом `(upgraded-array-element-type B)`.

Способ того, как обновляется тип элемента массива, зависит только от запрашиваемого типа элемента и не зависит от других свойств массива, таких как размер, ранг, расширяемость, наличия или отсутствия указателя заполнения, или относительности.

Обоснование: If upgrading were allowed to depend on any of these properties, all of which can be referred to, directly or indirectly, in the language of type specifiers, then it would not be possible to displace an array in a consistent and dependable manner to another array created with the same `:element-type` argument but differing in one of these properties.

Следует отметить, что `upgraded-array-element-type` может быть определён, как

```
(defun upgraded-array-element-type (type)
  (array-element-type (make-array 0 :element-type type)))
```

но, это определение и имеет недостаток в виде создания и удаления массива. Умная реализация конечно может имитировать создание для таких случаев.

[Функция] **upgraded-complex-part-type** *type*

Функция возвращает спецификатор типа, указывающий на тип наиболее приближенной для указанного типа *type* для частей

комплексного числа. Результат обязательно должен быть супертипом для переданного *type*. Кроме того, если тип *A* является подтипом *B*, тогда `(upgraded-complex-part-type A)` является подтипом `(upgraded-complex-part-type B)`.

Глава 5

Структура программы

В главе 2 был рассказано о синтаксисе записи Common Lisp'овых объектов. А так как все Common Lisp'овые программы также являются и объектами данных, то и синтаксис у них одинаковый.

Lisp'овые программы составляются из форм и функций. Формы *выполняются* (относительно некоторого контекста) для получения значений и побочных эффектов. Функции в свою очередь вызываются с некоторыми аргументами. Это называется помощью *применени* функции к аргументам. Наиболее важный вид форм выполняет вызов функции, и наоборот, функция выполняет вычисление с помощью выполнения форм.

В данной главе, сначала обсуждаются формы и затем функции. В конце, обсуждаются специальные формы «верхнего уровня (top level)». Наиболее важной из этих форм является `defun`, цель которой — создание именных функций (будут ещё и безымянные).

5.1 Формы

Стандартной единицей взаимодействия с реализацией Common Lisp'а является *форма*, которая является объектом данных, который выполняется как программа для вычисления одного или более *значений* (которые также являются объектами данных). Запросить выполнение можно для *любого* объекта данных, но не для всех это имеет смысл. Например, символы и списки имеет смысл выполнять, тогда как массивы обычно нет. Примеры содержательных форм: `3`, значение которой `3`, и

`(+ 3 4)`, значение которой `7`. Для обозначения этих фактов мы пишем `3` ⇒ `3` и `(+ 3 4)` ⇒ `7`. (⇒ означает «вычисляется в»)

Содержательные формы могут быть разделены на три категории: самовычисляемые формы, такие как числа, символы, которые используются для переменных, и списки. Списки в свою очередь могут быть разделены на три категории: специальные формы, вызовы макросов, вызовы функций.

Все стандартные объекты данных Common Lisp, не являющиеся символами и списками (включая `defstruct` структуры, определённые без опции `:type`) являются самовычисляемыми.

5.1.1 Самовычисляемые формы

Все числа, строковые символы, строки и битовые векторы являются *самовычисляемыми* формами. Когда данный объект вычисляется, тогда объект (или возможно копия в случае с числами и строковыми символами) возвращается в качестве значения данной формы. Пустой список (), который также является значением ложь (`nil`), также является самовычисляемой формой: значение `nil` является `nil`. Ключевые символы (примечание переводчика: не путать с ключевыми словами в других языках, в Common Lisp'е это вид символов) также вычисляются сами в себя: значение `:start` является `:start`.

Деструктивная модификация любого объекта, представленного как константа с помощью самовычисляемой формы или специальной формы `quote`, является ошибкой.

5.1.2 Переменные

В Common Lisp программах символы используются в качестве имён переменных. Когда символ вычисляется как форма, то в качестве результата возвращается значение переменной, которую данный символ именовал. Например, после выполнения `(setq items 3)`, которая присвоила значение `3` переменой именованной символом `items`, форма `items` выполнится в `3` (`items` ⇒ `3`). Переменные могут быть *назначены* с помощью `setq` или *связаны* с помощью `let`. Любая программная конструкция, которая связывает переменную, сохраняет старое значение переменной, и назначает новое, и при выходе из конструкции восстанавливается старое значение.

В Common Lisp'е есть два вида переменных. Они называются `лексические` (или *статические*) и *специальные* (или *динамические*). В одно время каждая из них или обе переменные с одинаковым именем могут иметь некоторое значение. На какую переменную ссылается символ при его вычислении, зависит от контекста выполнения. Главное правило заключается в том, что если символ вычисляется в тексте конструкции, которая создала *связывание* для переменной с одинаковым именем, то символ ссылается на переменную, обозначенную в этом связывании, если же в тексте такой конструкции нет, то символ ссылается на специальную переменную.

Различие между двумя видами переменных заключается в области видимости и продолжительности видимости. Лексически связанная переменная может быть использована *только* в тексте формы, которая установила связывание. Динамически связанная (специальная) переменная может быть использована в любое *время* между установкой связи и до выполнения конструкции, которая упраздняет связывание. Таким образом лексическое связывание переменных накладывает ограничение на использование переменной только в некоторой текстовой области (но не на временные ограничения, так связывание продолжает существовать, пока возможно существование ссылки на переменную). И наоборот, динамическое связывание переменных накладывает ограничение на временные рамки использования переменной (но не на текстовую область). Для более подробной информации смотрите главу 3.

Когда нет связываний, значение, которое имеет специальная переменная, называется *глобальным* значением (специальной) переменной. Глобальное значение может быть задано переменной только с помощью назначения, потому что значение заданное связыванием по определению не глобально.

Специальная переменная может вообще не иметь значения, в таком случае, говориться, что она *несвязанная*. По умолчанию, каждая глобальная переменная является несвязанной, пока значение не будет назначено явно, за исключением переменных определённых в этой книге или реализацией, которые уже имеют значения сразу после первого запуска Lisp машины. Кроме того, существует возможность установки связывания специальной переменной и затем упразднения этого связывания с помощью функции `makunbound`. В такой ситуации переменная также называется «несвязанной», хотя это и неправильно, если быть точнее, переменная связана, но без значения FIXME. Ссылка

на несвязанную переменную является ошибкой.

Чтение несвязанной переменной или неопределённой функции может быть обработано при самом высоком уровне безопасности (смотрите свойство `safety` декларации `optimize`). При других уровнях безопасности поведение не определено. Таким образом, чтение несвязанной переменной или неопределённой функции должно сигнализировать ошибку. («Чтение функции» включает ссылку на функцию используя специальный оператор `function`, как для `f` в форме `(function f)` и ссылку при вызове функции, как для `f` в форме `(f x y)`.)

В случае `inline` функций (в реализациях где они поддерживаются), выполнение встраивания функци представляет собой чтение функции, таким образом нет необходимости в проверке `fboundp` во время исполнения. Иными словами, результат применения `fmakunbound` к имени `inline` функции не определён.

При несвязанной переменой сигнализируется ошибка `unbound-variable`, и слот `name` условия `unbound-variable` содержит значение имени переменной, вызвавшей ошибку.

При неопределённой функции сигнализируется ошибка `unbound-function`, и слот `name` условия `unbound-function` содержит значение имени переменной, вызвавшей ошибку.

Тип условия `unbound-slot`, которое унаследовано от `cell-error`, имеет дополнительный слот `instance`, который может быть инициализировать параметром `:instance` в функции `make-condition`. Функция `unbound-slot-instance` предоставляет доступ к этому слоту.

Тип ошибки, по-умолчанию сигнализирующейся для метода обобщённой функции `slot-unbound` CLOS, является `unbound-slot`. Слот `instance` условия `unbound-slot` устанавливается в соответствующий экземпляр объекта и слот `name` в соответствующее имя переменной.

Некоторые глобальные переменные зарезервированы в качестве «именованных констант». Они имеют глобальное значение и не могут быть связаны или переназначены. Например символы `t` и `nil` зарезервированы. Этим символам невозможно назначить значение. Также и невозможно связать эти символы с другими значениями. Символы констант определённых с помощью `defconstant` также становятся зарезервированными и не могут быть переназначены или связаны (но они могут быть переопределены с помощью вызова `defconstant`). Ключевые символы также не могут быть переназначены

или связаны, ключевые символы всегда вычисляются сами в себя.

5.1.3 Специальные операторы

Если список выполняется в качестве формы, первым шагом является определение первого элемента списка. Если первый элемент списка является одним из символов, перечисленных в таблице 5.1, тогда список называется *специальным оператором*. (Использование слова «специальный» никак не связано с использованием этого слова в фразе «специальная переменная».)

Специальные операторы обычно являются окружениями и управляющими конструкциями. Каждый специальный оператор имеет свой идиосинкразический синтаксис. Например, специальный оператор `if`: `(if p (+ x 4) 5)` в Common Lisp'е означает то же, что и «**if** p **then** x+4 **else** 5» означает в Algol'е.

Выполнение специального оператора обычно возвращает одно или несколько значений, но выполнение может и вызвать нелокальный выход; смотрите `return-from`, `go` и `throw`.

Множество специальных операторов в Common Lisp'е фиксировано. Создание пользовательских специальных форм невозможно. Однако пользователь может создавать новые синтаксические конструкции с помощью определения макросов.

Множество специальных операторов в Common Lisp'е специально держится малым, потому что любая программа, анализирующая программы, должна содержать специальные знания о каждом типе специального оператора. Такие программы не нуждаются в специальных знаниях о макросах, так как раскрытие макроса просто, и далее остаётся только оперирование с результатом раскрытия. (Это не значит, что программы, в частности, компиляторы, не будут иметь специальных знаний о макросах. Компилятор может генерировать боле эффективный код, если он распознает такие конструкции, как `typecase` и `multiple-value-bind` и будет по-особому с ними обращаться.)

Реализация может исполнять в виде макроса любую конструкцию, описанную здесь как специальную оператор. И наоборот, реализация может выполнять в виде специального оператора любую конструкцию, описанную здесь как макрос, при условии, что также предоставляется эквивалентное определение макроса. Практическое значение заключается в том, что предикаты `macro-function` и

Таблица 5.1: Имена всех специальных форм

`block`	`if`	`progv`
`catch`	`labels`	`quote`
	`let`	`return-from`
`declare`	`let*`	`setq`
`eval-when`	`macrolet`	`tagbody`
`flet`	`multiple-value-call`	`the`
`function`	`multiple-value-prog1`	`throw`
`go`	`progn`	`unwind-protect`
		`symbol-macrolet`
	`locally`	`load-time-value`

`special-operator-p` могут оба возвращать истину, принимая один и тот же символ. Рекомендуется, чтобы программа для анализа других программ обрабатывала форму являющуюся списком с символом в первой позиции следующим образом:

1. Если программа имеет подробные знания о символе, обрабатывать форму необходимо с помощью специализированного кода. Все символы, перечисленные в таблице 5.1 должны попадать под данную категорию.

2. В противном случае, если для этого символа `macro-function` вычисляется в истину, необходимо применить `macroexpand` или `macroexpand-1` для раскрытия формы, и результат вновь анализировать.

3. В противном случае, необходимо расценивать форму как вызов функции.

5.1.4 Макросы

Если форма является списком и первый элемент не обозначает специальную форму, возможно он является именем *макроса*. Если так, то форма называется *макровызовом или вызовом макроса (macro-call)*. Макрос это функция, которая принимает формы и возвращает формы. Возвращённые формы подставляются в то место, где

происходил макровызов, и затем выполняются. (Этот процесс иногда называется *раскрытием макроса*.) Например, макрос с именем `return` принимает форму, вот так: `(return x)`, и полученная в результате раскрытия форма такая: `(return-from nil x)`. Мы говорим: старая форма раскрылась в новую. Новая форма будет вычислена на месте оригинальной формы. Значение новой формы будет возвращено, как значение оригинальной формы.

Макровызовы, и подформы макровызовов не обязательно должны быть Ъ списками, но использование списков с точкой требует совпадения с определением макроса, например «`. var`» или «`&rest var`». Таким образом определение макроса с точкой, позволяет корректно обрабатывать макровызовы и их подформы с точкой.

В Common Lisp'е существует некоторое количество стандартных макросов, и пользователь может определять свои макросы используя `defmacro`.

Макросы, предоставляемые реализацией Common Lisp'а и описанные здесь, могут раскрываться в код, который не будет являться переносимым между реализациями. Вызов макроса является портабельным, в то время как результат раскрытия нет.

Заметка для реализации: Implementors are encouraged to implement the macros defined in this book, as far as is possible, in such a way that the expansion will not contain any implementation-dependent special operators, nor contain as forms data objects that are not considered to be forms in Common Lisp. The purpose of this restriction is to ensure that the expansion can be processed by a program-analyzing program in an implementation-independent manner. There is no problem with a macro expansion containing calls to implementation-dependent functions. This restriction is not a requirement of Common Lisp; it is recognized that certain complex macros may be able to expand into significantly more efficient code in certain implementations by using implementation-dependent special operators in the macro expansion.

5.1.5 Вызовы функций

Если список выполняется как форма, и первый элемент не является символом, обозначающим специальную форму или макрос, тогда предполагается, что список является *вызовом функции*. Первый элемент списка является именем функции. Все следующие элементы списка будут вычислены. Одно значение каждого вычисленного элемента будет

является *аргументом* для вызываемой функции. Затем функция будет *применена* к аргументам. Вычисление функции обычно возвращает значение, однако вместо этого может быть выполнен нелокальный выход, смотрите `throw`. Функция может возвращать 0 и более значений, смотрите `values`. Если и когда функция возвращает значения, они становятся значениями вычисления формы вызова функции.

Например, рассмотрим вычисление формы: `(+ (* 4 5))`. Символ `+` обозначает функцию сложения, а не специальную форму или макрос. Таким образом две формы `3` и `(* 4 5)` вычисляются для аргументов. Форма `3` вычисляется в `3`, а форма `(* 4 5)` является вызовом функции (умножения). Таким образом формы `4` и `5` вычисляются сами в себя, тем самым предоставляя аргументы для функции умножения. Функция умножения вычисляет результат `20` и возвращает его. Значения `3` и `20` становятся аргументами функции сложения, которая вычисляет и возвращает результат `23`. Таким образом мы говорим `(+3 (* 4 5))` $\Rightarrow$ `23`.

Тогда как аргументы в вызове функции всегда выполняются слева направо, поиск самой функции для вызова может производится как до выполнения аргументов, так и после. Программы, которые полагаются на тот факт, что поиск функции осуществляется перед выполнением аргументов, являются ошибочными.

5.2 Функции

Существуют два метода указать функцию для использования в форме вызова функции. Один из них заключается в указании символа имени функции. Это использование символов для обозначения функций полностью независимо от их использования для обозначения специальных и лексических переменных. Другой путь заключается в использовании *лямбда-выражения*, которое является списком с первым элементом равным `lambda`. Лямбда-выражение *не* является формой, оно не может быть полноценно вычислено. Лямбда выражения и символы, когда они используются в программах для обозначения функций, могут быть указаны в качестве первого элемента формы вызова функции, или только в качестве второго параметры в специальной форме `function`. Следует отметить, что в этих двух контекстах символы и лямбда-выражения обрабатываются, как *имена* функций.

Необходимо отличать это от обработки символов и лямбда выражений, как *функциональных объектов, или объектов функций (function objects)*, которые удовлетворяют предикату `functionp`, как при представлении таких объектов в вызовы функций `apply` или `funcall`.

5.2.1 Именованные функции

Функция может иметь два типа имена. *Глобальное имя* может быть дано функции с помощью конструкции *defun*. *Локальное имя* может быть дано функции с помощью специальных форм `flet` или `labels`. Когда для функции задаётся имя, то с этим именем связывается лямбда-выражение с информацией о сущностях, которые были лексически доступны на момент связи. Если в качестве первого элементы формы вызова функции используется символ, тогда он ссылается на определение функции из наиболее ближней формы `flet` или `labels`, которые в своем тексте содержат эту форму, иначе символ ссылается на глобальное определение функции, при отсутствии вышеназванных форм.

5.2.2 Лямбда-выражения

Лямбда-выражение является списком со следующим синтаксисом:

(lambda *lambda-list . body*)

Первый элемент должен быть символом `lambda`. Второй элемент должен быть списком. Он называется *лямбда-списком*, и задаёт имена для *параметров* функции. Когда функция, обозначенная лямбда-выражением, применяется к аргументам, аргументы подставляются в соответствии с лямбда-списком. *body* может впоследствии ссылаться на аргументы используя имена параметров. *body* состоит из любого количества форм (возможно нулевого количества). Эти формы выполняются последовательно, и в качестве значения возвращается результат только *последней* формы (в случае отсутствия форм, возвращается `nil`). Полный синтаксис лямбда-выражения:

```
(lambda ( {var}*
        [&optional {var / (var [initform [svar]])}*]
        [&rest var]
        [&key {var / ({var / (keyword var)} [initform [svar]])}*          [&allow-oth
        [&aux {var / (var [initform])}*] )
  [[ {}*declaration / documentation-string]]
  {form}* )
```

Каждый элемент лямбда-списка является или спецификатором параметра или *ключевым символом лямбда-списка*. Ключевые символы лямбда-списка начинаются с символа `&`. Следует отметить, что ключевые символы лямбда-списка не является ключевыми символами в обычном понимании. Они не принадлежат пакету `keyword`. Они являются обычными символами, имена которых начинаются амперсандом. Такая терминология запутывает, но так сложилась история.

Keyword in the preceding specification of a lambda-list may be any symbol whatsoever, not just a keyword symbol in the `keyword` package. See below.

Keyword в предыдущем определении лябмда-списка может быть любым, а не только ключевым из пакета `keyword`, символом. Смотрите ниже.

Лямбда-список имеет пять частей, любая или все могут быть пустыми:

- Спецификаторы для *обязательных параметров*. К ним относятся все спецификаторы параметров до первого ключевого символа лямбда-списка. Если такой ключевой символ отсутствует, все спецификаторы считаются обязательными.

- Спецификаторы для *необязательных* параметров. Если указан ключевой символ `&optional`, после него будут следовать спецификаторы *необязательных* параметров вплоть до следующего ключевого слова, или до конца списка.

- Спецификатор для *неопределённого количества или оставшегося (rest)* параметра. Если указан ключевой символ `&rest`, за ним

должен следовать только один спецификатор *оставшегося (rest)* параметра, за которым может следовать другой ключевой символ или лямбда-список может заканчиваться.

- Спецификатор для *именованных (keyword)* параметров. Если указан ключевой символ `&key`, все спецификаторы после данного символа до следующего ключевого символа или конца списка являются спецификаторами *именованных* параметров. За спецификаторами именованных параметров опционально может использовать ключевой символ `&allow-others-keys`.

- Спецификатор для *вспомогательных (aux)* переменных. Они не являются параметрами. Если указан ключевой символ `&aux`, все спецификаторы после него являются спецификаторами вспомогательных переменных.

Когда функция, заданная лямбда-выражением, применяется к аргументам, то эти аргументы и параметры вычисляются слева направо. В простейшем случае, в лямбда-списке присутствуют только обязательные параметры. Каждый из них задаётся просто именем переменной *var* параметра. Когда функция применяется, аргументов должно быть столько же, сколько и параметров, и каждый параметр связывается с одним аргументом. В общем случае, каждый параметр связывается как лексическая переменная, если только с помощью декларации не указано, что связь должна осуществляться, как для специальной переменной. Смотрите `defvar`, `proclaim`, `declare`.

В более общем случае, если указано n обязательных параметров (n может равняться нулю), тогда должно быть как минимум n аргументов, и обязательные параметры будут связаны с n первыми аргументами.

Если указаны необязательные параметры, тогда каждый из них будет обработан так, как описано ниже. Если осталось некоторое количество аргументов, тогда переменная параметра `var` будет связана с оставшимся аргументом. Принцип такой же, как и для обязательных параметров. Если не осталось аргументов, тогда выполняется часть *initform*, и переменная параметра связывается с её результатом (или с `nil`, если форма *initform* не была задана). Если в спецификаторе указано имя ещё одной переменной *svar*, то она связывается с *true*, если аргумент был задан, и с *false* аргумент не был задан (и в таком случае выполнилась *initform*). Переменная *svar* называется *supplied-p*

параметр. Она связывается не с аргументом, а со значением, которое показывает был ли задан аргумент для данного параметра или нет.

После того, как все *необязательные* параметры были обработаны, может быть указан *оставшийся (rest)* параметр. Если *оставшийся (rest)* параметр указан, он будет связан со списком все оставшихся необработанных аргументов. Если таких аргументов не осталось, *оставшийся (rest)* параметр будет связан с пустым списком. Если в лямбда списке отсутствуют *оставшийся (rest)* параметр и *именованные (keyword)* параметры, то необработанных аргументов оставаться не должно (иначе будет ошибка).

X3J13 voted in January 1989 to clarify that if a function has a *rest* parameter and is called using `apply`, then the list to which the *rest* parameter is bound is permitted, but not required, to share top-level list structure with the list that was the last argument to `apply`. Programmers should be careful about performing side effects on the top-level list structure of a *rest* parameter.

This was the result of a rather long discussion within X3J13 and the wider Lisp community. To set it in its historical context, I must remark that in Lisp Machine Lisp the list to which a *rest* parameter was bound had only dynamic extent; this in conjunction with the technique of "cdr-coding" permitted a clever stack-allocation technique with very low overhead. However, the early designers of Common Lisp, after a great deal of debate, concluded that it was dangerous for cons cells to have dynamic extent; as an example, the "obvious" definition of the function `list`

```
(defun list (&rest x) x)
```

could fail catastrophically. Therefore the first edition simply implied that the list for a *rest* parameter, like all other lists, would have indefinite extent. This still left open the flip side of the question, namely, Is the list for a *rest* parameter guaranteed fresh? This is the question addressed by the X3J13 vote. If it is always freshly consed, then it is permissible to destroy it, for example by giving it to `nconc`. However, the requirement always to cons fresh lists could impose an unacceptable overhead in many implementations. The clarification approved by X3J13 specifies that the programmer may not rely on the list being fresh; if the function was called using `apply`, there is no way to know where the list came from.

Далее обрабатываются все *именованные (keyword)* параметры. Для этих параметров обрабатываются те же аргументы, что и для *оставшегося (rest)* параметра. Безусловно, возможно указывать и `&rest` и `&key`. В таком случае оставшиеся аргументы используются для обеих целей: все оставшиеся аргументы составляются в список для `&rest` параметра и они также обрабатываются, как `&key` параметры. Только в этой ситуации один аргумент может обрабатываться более чем для одного параметра. Если указан `&key`, должно остаться чётное количество аргументов. Они будут обработаны попарно. Первый аргумент в паре должен быть ключевым символом, который задаёт имя параметра, второй аргумент должен быть соответствующим значением.

A *keyword* in a lambda-list to be any symbol whatsoever, not just a keyword symbol in the `keyword` package. If, after `&key`, a variable appears alone or within only one set of parentheses (possibly with an *initform* and a *svar*), then the behavior is as before: a keyword symbol with the same name as the variable is used as the keyword-name when matching arguments to parameter specifiers. Only a parameter specifier of the form `((`*`keyword var`*`) ...)` can cause the keyword-name not to be a keyword symbol, by specifying a symbol not in the `keyword` package as the *keyword*. For example:

```
(defun wager (&key ((secret password) nil) amount)
  (format nil "You ~A $~D"
          (if (eq password 'joe-sent-me) "win" "lose")
          amount))

(wager :amount 100) ⇒ "You lose $100"
(wager :amount 100 'secret 'joe-sent-me) ⇒ "You win $100"
```

The `secret` word could be made even more secret in this example by placing it in some other `obscure` package, so that one would have to write

```
(wager :amount 100 'obscure:secret 'joe-sent-me) ⇒ "You win $100"
```

to win anything.

В каждом именованном параметре спецификатор должен быть назван *var* для переменной параметра. FIXME Если явно указан ключевой символ, тогда он будет использоваться для имени параметра.

В противном случае используется имя переменной `var` для поиска ключевого символа в аргументах. Таким образом:

(defun foo (`&key` radix (type 'integer)) ...)

означает то же, что и

(defun foo (`&key` ((:radix radix)) ((:type type) 'integer)) ...)

Спецификатор именованного (keyword) параметра, как и все спецификаторы параметров, обрабатывается слева направо. Для каждого спецификатора именованного параметра, если в паре аргумента, в которой ключевой символ совпадает с именем параметра (сравнение производится с помощью `eq`), тогда переменная параметра связывается значением из этой пары. Если имеется более одной пар аргументов с одинаковым именем, то это не ошибка. В таком случае используется наиболее левая пара. Если пары аргументов не нашлось, тогда выполняется *initform* и переменная параметра связывается с этим значением (или с `nil`, если *initform* не задана). Переменная *svar* используется в тех же целях, что и для *необязательных* параметров. Она будет связана с *истиной*, если была необходимая пара аргументов, и иначе — с *ложью*.

Если пара аргументов содержит ключевой символ, который не присутствует в спецификаторах параметров в лямбда списке, то или возникнет ошибка или возможны следующие условия:

- В лямбда-списке был указан `&allow-other-keys`.
- Где-то среди именованных аргументов есть пара, в которой есть ключевой символ `:allow-other-keys` и значение не равно `nil`.

В случае возникновения одного из этих условий, можно использовать именованные аргументы, которые не имеют соответствующих параметров (эти аргументы будут доступны, как оставшийся `&rest` параметр). Целью этого механизма является возможность объединять лямбда-списки разных функции без необходимости копировать все спецификаторы именованных (keyword) параметров. Например функция обёртка может передать часть именованных аргументов в

обернутую функцию без необходимости явного ручного указания их всех.

После того как все спецификаторы были обработаны, слева направо обрабатываются спецификаторы вспомогательных параметров. Для каждого из них выполняется *initform* и переменная *var* связывается с этим результатом (или с `nil`, если *initform* не определена). С `&aux` переменными можно делать то же, что и со специальной формой `let*`:

(lambda (x y &aux (a (car x)) (b 2) c) ...)
  ≡ (lambda (x y) (let* ((a (car x)) (b 2) c) ...))

Что использовать зависит только от стиля.

Когда какая-либо форма *initform* выполняется в каком-либо спецификаторе параметра, данная форма может ссылаться на любую переменную параметра, стоящую слева от данной формы, включая supplied-p переменные, и может рассчитывать на то, что другие переменные параметров ещё не связаны (включая переменную данного параметра).

После того как был обработан лямбда-список, выполняются формы из тела лямбда-выражения. Эти формы могут ссылаться на аргументы функции, используя имена параметров. При выходе из функции, как с помощью нормального возврата, так и с помощью нелокального выхода, связывания параметров, и лексические, и специальные, упраздняются. В случае создания «замыкания» над данными связываниями, связи упраздняются не сразу, а сначала сохраняются, чтобы потом быть вновь восстановленными.

Примеры использования `&optional` и `&rest` параметров:

((lambda (a b) (+ a (* b 3))) 4 5) ⇒ 19
((lambda (a `&optional` (b 2)) (+ a (* b 3))) 4 5) ⇒ 19
((lambda (a `&optional` (b 2)) (+ a (* b 3))) 4) ⇒ 10
((lambda (`&optional` (a 2 b) (c 3 d) `&rest` x) (list a b c d x)))
  ⇒ (2 `nil` 3 `nil` `nil`)
((lambda (`&optional` (a 2 b) (c 3 d) `&rest` x) (list a b c d x))
 6)
  ⇒ (6 t 3 `nil` `nil`)
((lambda (`&optional` (a 2 b) (c 3 d) `&rest` x) (list a b c d x))
 6 3)
  ⇒ (6 t 3 t `nil`)

```
((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x))
 6 3 8)
  ⇒ (6 t 3 t (8))
((lambda (&optional (a 2 b) (c 3 d) &rest x) (list a b c d x))
 6 3 8 9 10 11)
  ⇒ (6 t 3 t (8 9 10 11))
```

Примеры `&key` параметров:

```
((lambda (a b &key c d) (list a b c d)) 1 2)
  ⇒ (1 2 nil nil)
((lambda (a b &key c d) (list a b c d)) 1 2 :c 6)
  ⇒ (1 2 6 nil)
((lambda (a b &key c d) (list a b c d)) 1 2 :d 8)
  ⇒ (1 2 nil 8)
((lambda (a b &key c d) (list a b c d)) 1 2 :c 6 :d 8)
  ⇒ (1 2 6 8)
((lambda (a b &key c d) (list a b c d)) 1 2 :d 8 :c 6)
  ⇒ (1 2 6 8)
((lambda (a b &key c d) (list a b c d)) :a 1 :d 8 :c 6)
  ⇒ (:a 1 6 8)
((lambda (a b &key c d) (list a b c d)) :a :b :c :d)
  ⇒ (:a :b :d nil)
```

Пример смешения всех:

```
((lambda (a &optional (b 3) &rest x &key c (d a))
   (list a b c d x))
 1) ⇒ (1 3 nil 1 ())
```

```
((lambda (a &optional (b 3) &rest x &key c (d a))
   (list a b c d x))
 1 2) ⇒ (1 2 nil 1 ())
```

```
((lambda (a &optional (b 3) &rest x &key c (d a))
   (list a b c d x))
 :c 7) ⇒ (:c 7 nil :c ())

((lambda (a &optional (b 3) &rest x &key c (d a))
   (list a b c d x))
 1 6 :c 7) ⇒ (1 6 7 1 (:c 7))

((lambda (a &optional (b 3) &rest x &key c (d a))
   (list a b c d x))
 1 6 :d 8) ⇒ (1 6 nil 8 (:d 8))

((lambda (a &optional (b 3) &rest x &key c (d a))
   (list a b c d x))
 1 6 :d 8 :c 9 :d 10) ⇒ (1 6 9 8 (:d 8 :c 9 :d 10))
```

В лямбда-выражении, если оно стоит на первом месте в списке формы вызова функции, допускаются все ключевые символы лямбда-списка, хотя они и не очень-то полезны в таком контексте. Гораздо полезнее их использовать в глобальных функциях, определённых с помощью `defun`.

Все символы, что начинаются на `&` обычно зарезервированы для использования в качестве ключевых символов лямбда-списка, и не должны использоваться для имён переменных. Реализации Common Lisp'а могут также предоставлять свои дополнительные ключевые символы лямбда-списка.

[Константа] lambda-list-keywords

Значение `lambda-list-keywords` является списком всех ключевых символов лямбда-списка, используемых в данной реализации, включая те, которые используются только в `defmacro`. Этот список должен содержать как минимум символы `&optional`, `&rest`, `&key`, `&allow-other-keys`, `&aux`, `&body`, `&whole` и `&environment`.

Вот пример использования `&allow-other-keys` и `:allow-other-keys`, рассматривающий функцию, которая принимает два своих именованных аргумента и также дополнительные именованные аргументы, которые затем передаются `make-array`:

```
(defun array-of-strings (str dims &rest keyword-pairs
                         &key (start 0) end &allow-other-keys)
  (apply #'make-array dims
         :initial-element (subseq str start end)
         :allow-other-keys t
         keyword-pairs))
```

Такая функция принимает строку и информацию о размерности и возвращает массив с заданной размерностью, каждый из элементов которого равен заданной строке. Именованные аргументы `:start` и `:end`, как обычно (смотрите главу 14), можно использовать для указания того, что должна использоваться подстрока. Кроме того, использование `&allow-other-keys` в лямбда списке указывает на то, что вызов этой функции может содержать дополнительные именованные аргументы. Для доступа к ним используется `&rest` аргумент. Эти дополнительные именованные аргументы передаются в `make-array`. `make-array` не принимает именованные аргументы `:start` и `:end`, и было бы ошибкой допустить их использование. Однако указание `:allow-other-keys` равное не-`nil` значению позволяет передавать любые другие именованные аргументы, включая `:start` и `:end`, и они были бы приняты и проигнорированы.

[Константа] **lambda-parameters-limit**

Значение `lambda-parameters-limit` является положительным целым, которое невключительно является верхней границей допустимого количества имён параметров, которые могут использоваться в лямбда-списке. Значение зависит от реализации, но не может быть менее 50. Разработчики поощряются за создание данной границы как можно большей без потери производительности. Смотрите `call-arguments-list`.

5.3 Формы верхнего уровня

Стандартный путь взаимодействия с реализацией Common Lisp через *цикл чтение-выполнение-печать* (*read-eval-print loop*): система

циклично считывает форму из некоторого источника ввода (клавиатура, файл), выполняет её, затем выводит результат(ы) в некоторое устройство вывода (дисплей, другой файл). Допускается любая форма (выполняемый объект данных), однако существуют некоторые ормы разработанные для удобного применения в качестве форм *верхнего уровня*. Эти специальные формы верхнего уровня могут использоваться для определения глобальных функции (globally named functions), макросов, создания деклараций и определения глобальных значений для специальных переменных.

While defining forms normally appear at top level, it is meaningful to place them in non-top-level contexts. All defining forms that create functional objects from code appearing as argument forms must ensure that such argument forms refer to the enclosing lexical environment. Compilers must handle defining forms properly in all situations, not just top-level contexts. However, certain compile-time side effects of these defining forms are performed only when the defining forms occur at top level (see section 24.1).

Макросы обычно определяются с помощью специальной формы `defmacro`. Этот механизм достаточно сложен. Для подробностей смотрите главу 8.

5.3.1 Определение функций

Специальная форма `defun` служит для определения функций.

[Макрос] **defun** name lambda-list
[[{declaration}* | doc-string]]
{form}*

Выполнение формы `defun` приводит к тому, что символ *name* становиться глобальным именем для функции определённой лямбда-выражением.

(lambda *lambda-list* {declaration / doc-string}* {form}*)

определяется в лексическом окружении, в котором выполнялась форма `defun`. А так как формы `defun` обычно выполняются на самом верхнем уровне, лямбда-выражение обычно выполняется в нулевом лексическом окружении.

Тогда как данная форма обычно используется на самом верхем уровне, ее можно также использовать внутри других форм. `defun` может определять функцию внутри некоторого, не нулевого, лексического окружения.

`defun` в качестве параметра *name* принимает любое имя функции (символ или список, у которого *car* элемент равен `setf`—смотрите раздел 7.1). Так теперь для определения `setf`-оператора для функции `cadr` можно записать

```
(defun (setf cadr) ...)
```

Это удобнее, чем использование `defsetf` или `define-modify-macro`.

Если указана необязательная строка документации *doc-string*, тогда она присоединяется к символу *name* в качестве строки документации типа `function`. Смотрите `documentation`. Если после *doc-string* нет деклараций, строка документации может быть использована только при условии существования хотя бы одной формы после неё, иначе она будет использована в качестве форм *form* функции. Указывать более чем одну строку *doc-string* является ошибкой.

Формы *forms* составляют тело определяемой функции. Они выполняются как неявный `progn`.

Тело определяемой функции неявно заключается в конструкцию `block`, имя которой совпадает с *именем (name)* функции. Таким образом для выхода из функции может быть использовано выражение`return-from`.

В некоторых реализациях в `defun` могут также выполняться другие специальные учётные действия. *name* возвращается в качестве значения формы `defun`. Например:

```
(defun discriminant (a b c)
  (declare (number a b c))
  "Вычисляет дискриминант квадратного уравнения.
   Получает a, b и c и если они являются действительными числами (не комплексными),
    вычисляет значение b^2-4*a*c.
   Квадратное уравнение a*x^2+b*x+c=0 имеет действительные, multiple,
   или комплексные корни в зависимости от того, какое соответственно значение было по
   положительное, ноль или отрицательное."
  (- (* b b) (* 4 a c)))
```

⇒ discriminant
теперь (discriminant 1 2/3 -2) ⇒ 76/9

Пользователю разрешено использовать `defun` для переопределения функции, например, для установки корректной версии некорректного определения. Пользователю также разрешено переопределять макрос на функцию. Однако является ошибкой, попытка переопределить имя специальной формы (смотрите таблицу 5.1) на функцию.

5.3.2 Определение глобальных переменных и констант

Для определения глобальных переменных используются специальные формы `defvar` и `defparameter`. Для определения констант используется специальная форма `defconstant`.

[Макрос] **defvar** name [initial-value [documentation]]
[Макрос] **defparameter** name initial-value [documentation]
[Макрос] **defconstant** name initial-value [documentation]

`defvar` рекомендуется для декларации использования в программе специальных переменных.

(defvar *variable*)

указывает на то, что переменная *variable* будет специальной (`special`) (смотрите `proclaim`), и может выполнять некоторые учётные действия, зависимые от реализации.

Если *initial-value* не было указано, `defvar` не изменяет значение переменной *variable*. Если *initial-value* не было указано и переменная не имела значения, `defvar` не устанавливает значение.

Если для формы указан второй аргумент,

(defvar *variable initial-value*)

тогда переменная *variable*, если она ещё не была проинициализирована, инициализируется результатом выполнения формы *initial-value*. Форма *initial-value* не выполняется, если в этом нет необходимости. Это полезно, если форма *initial-value* выполняет что-то трудоёмкое, как, например, создание большой структуры данных.

Если не существует специального связывания этой переменной, инициализация производится присвоением глобального значения переменной. Обычно, такого связывания быть не должно. FIXME.

`defvar` также предоставляют хорошее место для комментария, описывающего значение переменной, тогда как обычное `special` указание соблазняет задекларировать несколько переменных за один раз и не предоставляет возможности прокомментировать их.

```
(defvar *visible-windows* 0
  "Количество видимых окон на экране")
```

`defparameter` подобна `defvar`, но `defparameter` требует обязательной формы *initial-value*, и, выполняя эту форму присваивает результат переменной. Семантическое различие заключается в том, что `defvar` предназначена декларировать переменную, изменяемую программой, тогда как `defparameter` предназначена для декларации переменной, как константы, которая может быть изменена (и во время выполнения программы), для изменения поведения программы. Таким образом `defparameter` не указывает, что количество никогда не изменяется, в частности, она не разрешает компилятору предположить то, что значение может быть вкомпилировано в программу.

`defconstant` похожа на `defparameter`, но в отличие от последней, указывает, что значение переменной *name* фиксировано и позволяет компилятору предположить, что значение может быть вкомпилировано в программу. Однако, если компилятор для оптимизации выбирает путь замены ссылок на имя константы на значения этой константы в компилируемом коде, он должен позаботиться о том, чтобы такие «копии» были эквивалентны `eql` объектам-значениям констант. Например, компилятор может спокойно копировать числа, но должен позаботиться об этом правиле, если значение константы является списком.

Если для переменной на момент вызова формы `defconstant` существует специальные связывания, то возникает ошибка (но реализации могут проверять, а могут и не проверять этот факт).

Если имя задекларировано с помощью `defconstant`, последующие присваивания и связывания данной специальной переменной будут являться ошибкой. Это справедливо для системозависимых констант, например, `t` и `most-positive-fixnum`. Компилятор может также сигнализировать о связывании лексической переменной с одинаковым именем.

Для любой из этих конструкций, документация должна быть строкой. Строка присоединяется к имени переменной, параметра или константы как тип документации `variable`, смотрите функцию `documentation`.

documentation-string не выполняется и должна представлять строку, когда выполняется `defvar`, `defparameter` или `defconstant`.

Например, форма

```
(defvar *avoid-registers* nil "Compilation control switch #43")
```

законна, но

```
(defvar *avoid-registers* nil
  (format nil "Compilation control switch #~D"
          (incf *compiler-switch-number*)))
```

ошибочна, так как вызов `format` не является дословно строкой.

С другой стороны, форма

```
(defvar *avoid-registers* nil
  #.(format nil "Compilation control switch #~D"
            (incf *compiler-switch-number*)))
```

может использоваться для вышеназванной цели, потому что вызов `format` выполняется на во время чтения кода `read`, когда форма `defvar` выполняется, в ней указана строка, которая являлась результатом вызова `format`.

Эти конструкции обычно используются только как формы верхнего уровня. Значения, возвращаемые каждой из этих конструкций, это задекларированные имена *name*.

5.3.3 Контроль времени выполнения

[Специальный оператор] **eval-when** ({situation}*) {form}*

Тело формы `eval-when` выполняется как неявный `progn`, но только в перечисленных ниже ситуациях. Каждая ситуация *situation* должна быть одним символов, `:compile-toplevel`, `:load-toplevel` или `:execute`.

Использование `:compile-toplevel` и `:load-toplevel` контролирует, что и когда выполняется для форм верхнего уровня. Использование `:execute` контролирует будет ли производится выполнения форм не верхнего уровня.

Конструкция `eval-when` может быть более понятна в терминах модели того, как компилятор файлов, `compile-file`, выполняет формы в файле для компиляции.

Формы следующие друг за другом читаются из файла с помощью компилятора файла используя `read`. Эти формы верхнего уровня обычно обрабатываются в том, что мы называем режим «времени некомпиляции (not-compile-time mode)». Существует и другой режим, называемый режим «времени-компиляции (compile-time-too mode)», которые вступает в игру для форм верхнего уровня. Специальная форма `eval-when` используется для аннотации программы таким образом, чтобы предоставить программе, осуществлять выбор режима.

Обработка форм верхнего уровня в компиляторе файла работает так, как рассказано ниже:

- Если форма является макровызовом, она разворачивается и результат обрабатывается, как форма верхнего уровня в том же режиме обработки (времени-компиляции или времени-некомпиляции, (compile-time-too или not-compile-time).
- Если форма `progn` (или `locally`), каждая из форм из их тел обрабатываются, как формы верхнего уровня в том же режиме обработки.
- Если форма `compiler-let`, `macrolet` или `symbol-macrolet`, компилятор файла создаёт соответствующие связывания и рекурсивно обрабатывает тела форм, как неявный `progn` верхнего уровня в контексте установленных связей в том же режиме обработки.

- Если форма `eval-when`, она обрабатывается в соответствии со следующей таблицей:

LT	CT	EX	CTTM	Действие
да	да	–	–	обработать тело в режиме время-компиляции
да	нет	да	да	обработать тело в режиме время-компиляции
да	нет	–	нет	обработать тело в режиме время-некомпиляции
да	нет	нет	–	обработать тело в режиме время-некомпиляции
нет	да	–	–	выполнить тело
нетнет		да	да	выполнить тело
нетнет		–	нет	ничего не делать
нетнет		нет	–	ничего не делать

  В этой таблице столбец LT спрашивает присутствует ли `:load-toplevel` в ситуациях указанных в форме `eval-when`. CT соответственно указывает на `:compile-toplevel` и EX на `:execute`. Столбец CTTM спрашивает встречается ли форма `eval-when` в режиме времени-компиляции. Фраза «обработка тела» означает обработку последовательно форм тела, как неявного `progn` верхнего уровня в указанном режиме, и «выполнение тела» означает выполнение форм тела последовательно, как неявный `progn` в динамическом контексте выполнения компилятора и в лексическом окружении, в котором встретилась `eval-when`.

- В противном случае, форма верхнего уровня, которая не представлена в специальных случаях. Если в режиме времени-компиляции, компилятор сначала выполняет форму и затем выполняет обычную обработку компилятором. Если установлен режим времени-некомпиляции, выполняется только обычная обработка компилятором (смотрите раздел 24.1). Любые подформы обрабатываются как формы не верхнего уровня.

Следует отметить, что формы верхнего уровня обрабатываются гарантированно в порядке, в котором они были перечислены в тексте в файле, и каждая форма верхнего уровня прочтённая компилятором обрабатывается перед тем, как будет прочтена следующая. Однако, порядок обработки (включая, в частности, раскрытие макросов) подформ, которые не являются формами верхнего уровня, не определён.

Для формы `eval-when`, которая не является формой верхнего уровня в компиляторе файлов (то есть либо в интерпретаторе, либо `compile`, либо в компиляторе файлов, но не на верхнем уровне), если указана ситуация `:execute`, тело формы обрабатывается как неявный `progn`. В противном случае, тело игнорируется и форма `eval-when` имеет значение `nil`.

Для сохранения обратной совместимости, *situation* может также быть `compile`, `load` или `eval`. Внутри формы верхнего уровня `eval-when`, они имеют значения `:compile-toplevel`, `:load-toplevel` и `:execute` соответственно. Однако их поведение не определено при использовании в `eval-when` не верхнего уровня.

Следующие правила являются логическим продолжением предыдущих определений:

- Никогда не случится так, чтобы выполнение одного `eval-when` выражения приведёт к выполнению тела более чем один раз.
- Старый ключевой символ `eval` был неправильно использован, потому что выполнение тела не нуждается в `eval`. Например, когда определение функции

  ```
  (defun foo () (eval-when (:execute) (print 'foo)))
  ```

  скомпилируется, вызов `print` должен быть скомпилирован, а не выполнен во время компиляции.
- Макросы, предназначенные для использования в качестве форм верхнего уровня, должны контролировать все побочные эффекты, которые будут сделаны формами в процессе развёртывания. Разворачиватель макроса сам по себе не должен порождать никаких побочных эффектов.

  ```
  (defmacro foo ()
    (really-foo)                    ;Неправильно
    `(really-foo))
  ```

```
(defmacro foo ()
  `(eval-when (:compile-toplevel
               :load-toplevel :execute)     ;Правильно
     (really-foo)))
```

Соблюдение этого правила будет значит, что такие макросы будут вести себя интуитивно понятно при вызовах в формах не верхнего уровня.

- Расположение связывания переменной окружённой `eval-when` захватывает связывание, потому что режим «время-компиляции» не может случиться (потому что `eval-when` не может быть формой верхнего уровня)

```
(let ((x 3))
  (eval-when (:compile-toplevel :load-toplevel :execute)
    (print x)))
```

выведет 3 во время выполнения (в данном случае загрузки) и не будет ничего выводить во время компиляции. Разворачивание `defun` и `defmacro` может быть выполнено в контексте `eval-when` и могут корректно захватывать лексическое окружение. Например, реализация может разворачивать форму `defun`, такую как:

```
(defun bar (x) (defun foo () (+ x 3)))
```

в

```
(progn (eval-when (:compile-toplevel)
         (compiler::notice-function 'bar '(x)))
       (eval-when (:load-toplevel :execute)
         (setf (symbol-function 'bar)
               #'(lambda (x)
                   (progn (eval-when (:compile-toplevel)
                            (compiler::notice-function 'foo
                                                       '()))
                          (eval-when (:load-toplevel :execute)
                            (setf (symbol-function 'foo)
                                  #'(lambda () (+ x 3)))))))))
```

которая по предыдущим правилам будет обработана также, как и

```
(progn (eval-when (:compile-toplevel)
         (compiler::notice-function 'bar '(x)))
       (eval-when (:load-toplevel :execute)
         (setf (symbol-function 'bar)
               #'(lambda (x)
                   (progn (eval-when (:load-toplevel :execute)
                            (setf (symbol-function 'foo)
                                  #'(lambda () (+ x 3)))))))))
```

Вот несколько дополнительных примеров.

```
(let ((x 1))
  (eval-when (:execute :load-toplevel :compile-toplevel)
    (setf (symbol-function 'foo1) #'(lambda () x))))
```

`eval-when` в предыдущем выражении не является формой верхнего уровня, таким образом во внимание берётся только ключевой символ `:execute`. это не будет иметь эффекта во время компиляции. Однако этот код установит в `(symbol-function 'foo1)` функцию которая возвращает `1` во время загрузки (если `let` форма верхнего уровня) или во время выполнения (если форма `let` вложена в какую-либо другую форму, которая ещё не была выполнена).

```
(eval-when (:execute :load-toplevel :compile-toplevel)
  (let ((x 2))
    (eval-when (:execute :load-toplevel :compile-toplevel)
      (setf (symbol-function 'foo2) #'(lambda () x)))))
```

Если предыдущее выражение находилось на верхнем уровне в компилируемом файле, оно будет выполнятся в обоих случаях, и во время компиляции и во время загрузки.

```
(eval-when (:execute :load-toplevel :compile-toplevel)
  (setf (symbol-function 'foo3) #'(lambda () 3)))
```

Если предыдущее выражение находилось на верхнем уровне в компилируемом файле, оно будет выполняться в обоих случаях, и во время компиляции и во время загрузки.

```
(eval-when (:compile-toplevel)
  (eval-when (:compile-toplevel)
    (print 'foo4)))
```

Предыдущее выражение ничего не делает, оно просто возвращает `nil`.

```
(eval-when (:compile-toplevel)
  (eval-when (:execute)
    (print 'foo5)))
```

Если предыдущее выражение находилось на верхнем уровне в компилируемом файле, `foo5` будет выведено во время компиляции. Если эта форма была не на верхнем уровне, ничего не будет выведено во время компиляции. Вне зависимости от контекста, ничего не будет выведено во время загрузки или выполнения.

```
(eval-when (:execute :load-toplevel)
  (eval-when (:compile-toplevel)
    (print 'foo6)))
```

Если предыдущая форма находилась на верхнем уровне в компилируемом файле, `foo6` будет выведено во время компиляции. Если форма была не на верхнем уровне, ничего не будет выведено во время компиляции. Вне зависимости от контекста, ничего не будет выведение во время загрузки или выполнения кода.

Глава 6

Предикаты

Предикат — это функция, которая проверяет некоторое условие переданное в аргументах и возвращает `nil`, если условие ложное, или не-`nil`, если условие истинное. Можно рассматривать, что предикат производит булево значение, где `nil` обозначает *ложь* и все остальное — *истину*. Условные управляющие структуры, такие как `cond`, `if`, `when` и `unless` осуществляют проверку таких булевых значений. Мы говорим, что предикат *истинен*, когда он возвращает не-`nil` значение, и *ложен*, когда он возвращает `nil`, то есть он истинен или ложен в зависимости от того, истинно или ложно проверяемое условие.

По соглашению, имена предикатов обычно заканчиваются на букву `p` (которая обозначает «предикат (predicate)»). Common Lisp использует единое соглашение для использования дефисов в именах предикатов. Если имя предиката создано с помощью добавления `p` к уже существующему имени, такому как имя типа данных, тогда дефис помещается перед последним `p` тогда и только тогда, когда в исходном имени были дефисы. Например, `number` становится `numberp`, но `standard-char` становится `standard-char-p`. С другой стороны, если имя предиката сформировано добавлением префиксного спецификатора в начало существующего имени предиката, то два имени соединяются с помощью дефиса, и наличие или отсутствие перед завершающим `p` не изменяется. Например, предикат `string-lessp` не содержит дефиса перед `p`, потому что это строковая версия `lessp`. Имя `string-less-p` было бы некорректно указывающим на то, что это предикат проверяющий тип объекта называемого `string-less`, а имя `stringlessp` имело бы смысл того, что проверяет отсутствие строк в

чем-либо.

Управляющие структуры, которые проверяют булевы значения, проверяют только является или нет значение ложью (`nil`). Любое другое значение рассматривается как истинное. Часто предикат будет возвращать `nil`, в случае «неудачи» и некоторое *полезное* значение в случае «успеха». Такие функции могут использоваться не только для проверки, но и также для использования полезного значения, получаемого в случае успеха. Например `member`.

Если лучшего, чем не-`nil` значения, в целях указания успеха не оказалось, по соглашению в качестве «стандартного» значения истины используется символ `t`.

6.1 Логические значения

Имена `nil` и `t` в Common Lisp'е являются константами. Несмотря на то, что они являются обычными символами, и могут использоваться в качестве переменных при вычислениях, их значения не могут быть изменены. Смотрите `defconstant`.

[Константа] **nil**

Значение `nil` всегда `nil`. Этот объект обозначает логическую ложь, а также пустой список. Он также может быть записан, как `()`.

[Константа] `t`

Значение `t` всегда`t`.

6.2 Предикаты типов данных

Возможно наиболее важными предикатами в Lisp'е это предикаты, которые различают типы данных. То есть позволяют узнать принадлежит ли заданный объект данному типу. Также предикаты могут сравнивать два спецификатора типов.

6.2.1 Основные предикаты

Если тип данных рассматривать, как множество все объектов, принадлежащих этому типу, тогда функция `typep` проверяет принадлежность множеству, тогда как `subtypep` — принадлежность подмножеству.

[Функция] **typep** *object type*

`typep` является предикатом, который истинен, если объект *object* принадлежит типу *type*, и ложен в противном случае. Следует отметить, что объект может принадлежать нескольким типам, так как один тип может включать другой. *type* может быть любым спецификатором типа, описанным в главе 4, за исключением того, что он не может быть или включать список спецификатор типа, у которого первый элемент равен `function` или `values`. Спецификатор формы `(satisfies` *`fn`*`)` обрабатывается просто как применение функции *fn* к объекту *object* (смотрите `funcall`). Объект *object* принадлежит заданному типу, если результат не равен `nil`.

X3J13 voted in January 1989 to change `typep` to give specialized `array` and `complex` type specifiers the same meaning for purposes of type discrimination as they have for declaration purposes. Of course, this also applies to such type specifiers as `vector` and `simple-array` (see section 4.5). Thus

(typep foo '(array bignum))

in the first edition asked the question, Is `foo` an array specialized to hold bignums? but under the new interpretation asks the question, Could the array `foo` have resulted from giving `bignum` as the `:element-type` argument to `make-array`?

[Функция] **subtypep** *type1 type2*

Аргументы должны быть спецификаторами типов, но только теми, которые могут использоваться и для `typep`. Два спецификатора типа сравниваются. Данный предикат истинен, если тип *type1* точно является подтипом типа *type2*, иначе предикат ложен. Если результат `nil`, тогда тип *type1* может быть, а может и не быть подтипом типа *type2* (иногда это невозможно определить, особенно когда используется тип

`satisfies`). Второе возвращаемое значение указывает на точность результата. Если оно является истиной, значит первое значение указывает на точную принадлежность типов. Таким образом возможны следующие комбинации результатов:

`t`	`t`	*type1* точно является подтипом *type2*
`nil`	`t`	*type1* точно не является подтипом *type2*
`nil`	`nil`	`subtypep` не может определить отношение

X3J13 voted in January 1989 to place certain requirements upon the implementation of `subtypep`, for it noted that implementations in many cases simply "give up" and return the two values `nil` and `nil` when in fact it would have been possible to determine the relationship between the given types. The requirements are as follows, where it is understood that a type specifier *s* *involves* a type specifier *u* if either *s* contains an occurrence of *u* directly or *s* contains a type specifier *w* defined by `deftype` whose expansion involves *u*.

- `subtypep` is not permitted to return a second value of `nil` unless one or both of its arguments involves `satisfies`, `and`, `or`, `not`, or `member`.
- `subtypep` should signal an error when one or both of its arguments involves `values` or the list form of the `function` type specifier.
- `subtypep` must always return the two values `t` and `t` in the case where its arguments, after expansion of specifiers defined by `deftype`, are `equal`.

In addition, X3J13 voted to clarify that in some cases the relationships between types as reflected by `subtypep` may be implementation-specific. For example, in an implementation supporting only one type of floating-point number, `(subtypep 'float 'long-float)` would return `t` and `t`, since the two types would be identical.

Note that `satisfies` is an exception because relationships between types involving `satisfies` are undecidable in general, but (as X3J13 noted) `and`, `or`, `not`, and `member` are merely very messy to deal with. In all likelihood these will not be addressed unless and until someone is willing to write a careful specification that covers all the cases for the processing of these type specifiers by `subtypep`. The requirements stated above were easy to state and probably suffice for most cases of interest.

X3J13 voted in January 1989 to change `subtypep` to give specialized `array` and `complex` type specifiers the same meaning for purposes of type discrimination as they have for declaration purposes. Of course, this also applies to such type specifiers as `vector` and `simple-array` (see section 4.5).

If *A* and *B* are type specifiers (other than `*`, which technically is not a type specifier anyway), then `(array` *A*`)` and `(array` *B*`)` represent the same type in a given implementation if and only if they denote arrays of the same specialized representation in that implementation; otherwise they are disjoint. To put it another way, they represent the same type if and only if `(upgraded-array-element-type '`*A*`)` and `(upgraded-array-element-type '`*B*`)` are the same type. Therefore

(subtypep '(array *A*) '(array *B*))

is true if and only if `(upgraded-array-element-type '`*A*`)` is the same type as `(upgraded-array-element-type '`*B*`)`.

The `complex` type specifier is treated in a similar but subtly different manner. If *A* and *B* are two type specifiers (but not `*`, which technically is not a type specifier anyway), then `(complex` *A*`)` and `(complex` *B*`)` represent the same type in a given implementation if and only if they refer to complex numbers of the same specialized representation in that implementation; otherwise they are disjoint. Note, however, that there is no function called `make-complex` that allows one to specify a particular element type (then to be upgraded); instead, one must describe specialized complex numbers in terms of the actual types of the parts from which they were constructed. There is no number of type (or rather, *representation*) `float` as such; there are only numbers of type `single-float`, numbers of type `double-float`, and so on. Therefore we want `(complex single-float)` to be a subtype of `(complex float)`.

The rule, then, is that `(complex` *A*`)` and `(complex` *B*`)` represent the same type (and otherwise are disjoint) in a given implementation if and only if *either* the type *A* is a subtype of *B*, *or* `(upgraded-complex-part-type '`*A*`)` and `(upgraded-complex-part-type '`*B*`)` are the same type. In the latter case `(complex` *A*`)` and `(complex` *B*`)` in fact refer to the same specialized representation. Therefore

(subtypep '(complex *A*) '(complex *B*))

is true if and only if the results of `(upgraded-complex-part-type 'A)` and `(upgraded-complex-part-type 'B)` are the same type.

Under this interpretation

(subtypep '(complex single-float) '(complex float))

must be true in all implementations; but

(subtypep '(array single-float) '(array float))

is true only in implementations that do not have a specialized array representation for `single-float` elements distinct from that for `float` elements in general.

6.2.2 Специальные предикаты

Следующие предикаты осуществляют проверку определённых типов данных.

[Функция] **null** *object*

`null` истинен, если аргумент является (), иначе является ложью. Похожая операция производится `not`, однако `not` используется для отрицания булевых значение, тогда как `null` используется для проверки того, пустой ли список. Таким образом программист может выразить свои намерения, выбрав нужное имя функции.

(null x) ≡ (typep x 'null) ≡ (eq x '())

[Функция] **symbolp** *object*

`symbolp` истинен, если её аргумент является символом, в противном случае ложен.

(symbolp x) ≡ (typep x 'symbol)

[Функция] **atom** *object*

Предикат `atom` истинен, если аргумент не является cons-ячейкой, в противном случае ложен. Следует отметить `(atom '())` являет истиной, потому что () ≡ `nil`.

(atom x) ≡ (typep x 'atom) ≡ (not (typep x 'cons))

[Функция] **consp** *object*

Предикат `consp` истинен, если его аргумент является cons-ячейкой, в противном случае ложен. Следует отметить, пустой список не является cons-ячейкой, так `(consp '())` ≡ `(consp 'nil)` ⇒ `nil`.

(consp x) ≡ (typep x 'cons) ≡ (not (typep x 'atom))

[Функция] **listp** *object*

`listp` истинен, если его аргумент является cons-ячейкой или пустым списком (), в противном случае ложен. Она не проверяет является ли «список Ъ (true list)» (завершающийся `nil`) или «с точкой (dotted)» (завершающийся не-null атомом).

(listp x) ≡ (typep x 'list) ≡ (typep x '(or cons null))

[Функция] **numberp** *object*

`numberp` истинен, если аргумент это любой вид числа, в противном случае ложен.

(numberp x) ≡ (typep x 'number)

[Функция] **integerp** *object*

integerp истинен, если аргумент целое число, в противном случае ложен.

(integerp x) ≡ (typep x 'integer)

[Функция] **rationalp** *object*

`rationalp` истинен, если аргумент рациональное число (дробь или целое), в противном случае ложен.

(rationalp x) ≡ (typep x 'rational)

[Функция] **floatp** *object*

`floatp` истинен, если аргумент число с плавающей точкой, в противном случае ложен.

(floatp x) ≡ (typep x 'float)

[Функция] **realp** *object*

`realp` истинна, если аргумент является действительным числом, иначе ложна.

(realp x) ≡ (typep x 'real)

[Функция] **complexp** *object*

complexp истинен, если аргумент комплексное число, в противном случае ложен.

(complexp x) ≡ (typep x 'complex)

[Функция] **characterp** *object*

`characterp` истинен, если аргумент строковый символ, иначе ложен.

(characterp x) ≡ (typep x 'character)

[Функция] **stringp** *object*

`stringp` истинен, если аргумент строка, иначе ложен.

(stringp x) ≡ (typep x 'string)

[Функция] **bit-vector-p** *object*

`bit-vector-p` истинен, если аргумент битовый вектор, иначе ложен.

(bit-vector-p x) ≡ (typep x 'bit-vector)

[Функция] **vectorp** *object*

`vectorp` истинен, если аргумент вектор, иначе ложен.

(vectorp x) ≡ (typep x 'vector)

[Функция] **simple-vector-p** *object*

`vectorp` истинен, если аргумент простой общий вектор, иначе ложен.

(simple-vector-p x) ≡ (typep x 'simple-vector)

[Функция] **simple-string-p** *object*

`simple-string-p` истинен, если аргумент простая строка, иначе ложен.

(simple-string-p x) ≡ (typep x 'simple-string)

[Функция] **simple-bit-vector-p** *object*

`simple-bit-vector-p` истинен, если аргумент простой битовый вектор, иначе ложен.

(simple-bit-vector-p x) ≡ (typep x 'simple-bit-vector)

[Функция] **arrayp** *object*

`arrayp` истинен, если аргумент массив, иначе ложен.

(arrayp x) ≡ (typep x 'array)

[Функция] **packagep** *object*

`packagep` истинен, если аргумент является пакетом, иначе является ложью.

(packagep x) ≡ (typep x 'package)

[Функция] **functionp** *object*

(functionp x) ≡ (typep x 'function)

Типы `cons` и `symbol` непересекаются с типом `function`. `functionp` является ложной для символов и списков.

[Функция] **compiled-function-p** *object*

`compiled-function-p` истинен, если аргумент — скомпилированный объект кода, иначе ложен.

(compiled-function-p x) ≡ (typep x 'compiled-function)

Смотрите также `standard-char-p`, `string-char-p`, `streamp`, `random-state-p`, `readtablep`, `hash-table-p` и `pathnamep`.

6.3 Предикаты равенства

Common Lisp предоставляет ряд предикатов для проверки равенства двух объектов: `eq` (наиболее частный), `eql`, `equal` и `equalp` (наиболее общий). `eq` и `equal` имеют значения традиционные в Lisp'е. `eql` был добавлен, потому что он часто бывает необходим, и `equalp` был добавлен преимущественно, как версия `equal`, которая игнорирует различия типов при сравнении двух чисел и различия регистров при сравнении строковых символов. Если два объекта удовлетворяют любому из этих предикатов, то они также удовлетворяют всем тем, которые носят более общий характер.

[Функция] **eq** *x y*

`(eq x y)` является истиной тогда и только тогда, когда, *x* и *y* являются идентичными объектами. (В реализациях, *x* и *y* обычно равны `eq` тогда и только тогда, когда обращаются к одной ячейке памяти.)

Необходимо отметить, что вещи, которые выводят одно и то же, необязательно равны `eql` друг другу. Символы с одинаковым именем обычно равны `eq` друг другу, потому что используется функция `intern`. Однако, одинаковые значения чисел могут быть не равны `eq`, и два похожих списка обычно не равны `eq`. Например:

(eq 'a 'b) ложь
(eq 'a 'a) истина
(eq 3 3) может быть истина или ложь, в зависимости от реализации
(eq 3 3.0) ложь
(eq 3.0 3.0) может быть истина или ложь, в зависимости от реализации
(eq #c(3 -4) #c(3 -4))
может быть истина или ложь, в зависимости от реализации
(eq #c(3 -4.0) #c(3 -4)) ложь
(eq (cons 'a 'b) (cons 'a 'c)) ложь
(eq (cons 'a 'b) (cons 'a 'b)) ложь
(eq '(a . b) '(a . b)) может быть истина или ложь
(progn (setq x (cons 'a 'b)) (eq x x)) истина
(progn (setq x '(a . b)) (eq x x)) истина
(eq #\A #\A) может быть истина или ложь, в зависимости от реализации
(eq "Foo" "Foo") может быть истина или ложь
(eq "Foo" (copy-seq "Foo")) ложь
(eq "FOO" "foo") ложь

В Common Lisp'е, в отличие от других диалектов, реализация в любое время может создавать «копии» строковых символов и чисел. (Это сделано для возможности в повышении производительности.) Из этого следует правило, что Common Lisp не гарантирует для строковых символов и чисел то, что `eq` будет истинен, когда оба аргумента являются «одним и тем же». Например:

(let ((x 5)) (eq x x)) может быть истиной или ложью

Предикат `eql` означает то же, что и `eq`, за исключением того, что если аргументы являются строковыми символами или числами одинакового типа, тогда сравниваются их значения. Таким образом `eql` говорит, являются ли два объекта «концептуально (conceptually)» одинаковыми, тогда как `eq` указывает, являются ли два объекта «реализационно (implementationally)» одинаковыми. По этой причине сравнительным предикатом для функций работы с последовательностями, описанными в главе 14, является `eql`, а не `eq`.

Заметка для реализации: `eq` simply compares the two given pointers, so any kind of object that is represented in an "immediate" fashion will indeed have like-valued instances satisfy `eq`. In some implementations, for example, fixnums and characters happen to "work." However, no program should depend on this, as other implementations of Common Lisp might not use an immediate representation for these data types.

An additional problem with `eq` is that the implementation is permitted to "collapse" constants (or portions thereof) appearing in code to be compiled if they are `equal`. An object is considered to be a constant in code to be compiled if it is a self-evaluating form or is contained in a `quote` form. This is why `(eq "Foo" "Foo")` might be true or false; in interpreted code it would normally be false, because reading in the form `(eq "Foo" "Foo")` would construct distinct strings for the two arguments to `eq`, but the compiler might choose to use the same identical string or two distinct copies as the two arguments in the call to `eq`. Similarly, `(eq '(a . b) '(a . b))` might be true or false, depending on whether the constant conses appearing in the `quote` forms were collapsed by the compiler. However, `(eq (cons 'a 'b) (cons 'a 'b))` is always false, because every distinct call to the `cons` function necessarily produces a new and distinct cons.

X3J13 voted in March 1989 to clarify that `eval` and `compile` are not permitted either to copy or to coalesce ("collapse") constants (see `eq`) appearing in the code they process; the resulting program behavior must refer to objects that are `eql` to the corresponding objects in the source code. Only the `compile-file`/`load` process is permitted to copy or coalesce constants (see section 24.1).

[Функция] **eql** *x y*

Предикат `eql` истинен, если его аргументы равны `eq`, или если это

числа одинакового типа и с одинаковыми значениями, или если это одинаковые строковые символы. Например:

(eql 'a 'b) ложь
(eql 'a 'a) истина
(eql 3 3) истина
(eql 3 3.0) ложь
(eql 3.0 3.0) истина
(eql #c(3 -4) #c(3 -4)) истина
(eql #c(3 -4.0) #c(3 -4)) ложь
(eql (cons 'a 'b) (cons 'a 'c)) ложь
(eql (cons 'a 'b) (cons 'a 'b)) ложь
(eql '(a . b) '(a . b)) может быть истиной или ложью
(progn (setq x (cons 'a 'b)) (eql x x)) истина
(progn (setq x '(a . b)) (eql x x)) истина
(eql #\A #\A) истина
(eql "Foo" "Foo") может быть истиной или ложью
(eql "Foo" (copy-seq "Foo")) ложь
(eql "FOO" "foo") ложь

Обычно `(eql 1.0s0 1.0d0)` будет ложью, так как `1.0s0` и `1.0d0` не принадлежат одному типу данных. Однако в реализация может отсутствовать полный набор чисел с плавающей точкой, поэтому в такой ситуации `(eql 1.0s0 1.0d0)` может быть истиной. Предикат `=` будет сравнивать значения двух чисел, даже если числа принадлежат разным типам.

Если реализация поддерживает положительный и отрицательный нули, как различные значения (так IEEE стандарт предлагает реализовывать формат числа с плавающей точкой), тогда `(eql 0.0 -0.0)` будет ложью. В противном случае, когда синтаксис `-0.0` интерпретируется, как значение `0.0`, тогда `(eql 0.0 -0.0)` будет истиной. Предикат `=` отличается от `eql` в том, что `(= 0.0 -0.0)` будет всегда истинно, потому что `=` сравнивает математические значения операндов, тогда как `eql` сравнивает, так сказать, репрезентативные (representational) значения. FIXME.

Два комплексных числа будут равны `eql`, если их действительные части равны `eql` и мнимые части равны `eql`. Например, `(eql #C(4`

5) #C(4 5)) является истиной и (eql #C(4 5) #C(4.0 5.0)) является ложью. Следует отметить, что (eql #C(5.0 0.0) 5.0) ложь, а (eql #C(5 0) 5) истина. В случае с (eql #C(5.0 0.0) 5.0) два аргумента принадлежат разным типам и не равны eql, Однако, в случае (eql #C(5 0) 5), #C(5 0) не является комплексным числом, и автоматически преобразуется, по правилу канонизации комплексных чисел, в целое 5, так как дробное число 20/4 всегда упрощается до 5.

Случай (eql "Foo" "Foo") обсуждался выше в описании eq. Тогда как eql сравнивает значения чисел и строковых символов, он не сравнивает содержимое строк. Сравнение символов двух строк может быть выполнено с помощью equal, equalp, string= или string-equal.

[Функция] **equal** *x y*

Предикат equal истинен, если его аргументы это структурно похожие (изоморфные) объекты. Грубое правило такое, что два объекта равны equal тогда и только тогда, когда одинаково их выводимое представление.

Числа и строковые символы сравниваются также как и в eql. Символы сравниваются как в eq. Этот метод сравнения символов может нарушать правило и сравнении выводимого представления, в случае если различия двух символов с одинаковым выводимым представлением.

Объекты, которые содержат другие элементы, будут равны equal, если они принадлежат одному типу и содержащиеся элементы равны equal. Эта проверка реализована в рекурсивном стиле и может быть зациклиться на закольцованных структурах.

Для cons-ячеек, equal определён рекурсивно, как сравнение equal сначала *car* элементов, а затем *cdr*.

Два массива равны equal только, если они равны eq, с одним исключением: строки и битовые вектора сравниваются поэлементно. Если какой-либо аргумент или оба содержат указатель заполнения (fill pointer), данный указатель ограничит количество проверяемых с помощью equal элементов. Буквы верхнего и нижнего регистров в строках расцениваются предикатом equal как разные. (А equalp игнорирует различие в регистрах в строках.)

Два объекта имени файла (pathname objects) равны equal тогда и только тогда, когда все элементы (хост, устройство, и т.д.) равны. (Будут ли равны буквы разных регистров зависит от файловой системы.)

Имена файлов, которые равны `equal`, должны быть функционально эквивалентны.

`equal` рекурсивно рассматривает только следующие типы данных: cons-ячейки, битовые вектора, строки и имена файлов. Числа и строковые символы сравниваются так, как если бы сравнивались с помощью `eql`, а все остальные типы данных сравниваются как если бы с помощью `eq`.

(equal 'a 'b) ложь
(equal 'a 'a) истина
(equal 3 3) истина
(equal 3 3.0) ложь
(equal 3.0 3.0) истина
(equal #c(3 -4) #c(3 -4)) истина
(equal #c(3 -4.0) #c(3 -4)) ложь
(equal (cons 'a 'b) (cons 'a 'c)) ложь
(equal (cons 'a 'b) (cons 'a 'b)) истина
(equal '(a . b) '(a . b)) истина
(progn (setq x (cons 'a 'b)) (equal x x)) истина
(progn (setq x '(a . b)) (equal x x)) истина
(equal #\A #\A) истина
(equal "Foo" "Foo") истина
(equal "Foo" (copy-seq "Foo")) истина
(equal "FOO" "foo") ложь

Для сравнения дерева cons-ячеек применяя `eql` (или любой другой желаемый предикат) для листьев, используйте `tree-equal`.

[Функция] **equalp** *x y*

Два объекта равны `equalp`, если они равны `equal`, если они строковые символы и удовлетворяют предикату `char-equal`, который игнорирует регистр и другие атрибуты символов, если они числа и имеют одинаковое значение, даже если числа разных типов, если они включает в себя элементы, которые также равны `equalp`.

Объекты, которые включают в себя элементы, равны `equalp`, если они принадлежат одному типу и содержащиеся элементы равны `equalp`.

Проверка осуществляется в рекурсивном стиле и может не завершится на закольцованных структурах. Для cons-ячеек, предикат `equalp` определён рекурсивно и сравнивает сначала *car* элементы, а затем *cdr*.

Два массива равны `equalp` тогда и только тогда, когда они имеют одинаковое количество измерений, и размеры измерений совпадают, и все элементы равны `equalp`. Специализация массива не сравнивается. Например, строка и общий массив, случилось так, имеют одинаковые строковые символы, тогда они будут равны `equalp` (но определённо не равны `equal`). Если какой-либо аргумент содержит указатель заполнения, этот указатель ограничивает число сравниваемых элементов. Так как `equalp` сравнивает строки побуквенно, и не различает разных регистров букв, то сравнение строк регистронезависимо.

Два символа могут быть равны `equalp` только тогда, когда они `eq`, т.е. являются идентичными объектами.

X3J13 voted in June 1989 to specify that `equalp` compares components of hash tables (see below), and to clarify that otherwise `equalp` never recursively descends any structure or data type other than the ones explicitly described above: conses, arrays (including bit-vectors and strings), and pathnames. Numbers are compared for numerical equality (see `=`), characters are compared as if by `char-equal`, and all other data objects are compared as if by `eq`.

Two hash tables are considered the same by `equalp` if and only if they satisfy a four-part test:

- They must be of the same kind; that is, equivalent `:test` arguments were given to `make-hash-table` when the two hash tables were created.
- They must have the same number of entries (see `hash-table-count`).
- For every entry (*key1*, *value1*) in one hash table there must be a corresponding entry (*key2*, *value2*) in the other, such that *key1* and *key2* are considered to be the same by the `:test` function associated with the hash tables.
- For every entry (*key1*, *value1*) in one hash table and its corresponding entry (*key2*, *value2*) in the other, such that *key1* and *key2* are the same, `equalp` must be true of *value1* and *value2*.

The four parts of this test are carried out in the order shown, and if some part of the test fails, `equalp` returns `nil` and the other parts of the test are not attempted.

If `equalp` must compare two structures and the `defstruct` definition for one used the `:type` option and the other did not, then `equalp` returns `nil`.

If `equalp` must compare two structures and neither `defstruct` definition used the `:type` option, then `equalp` returns `t` if and only if the structures have the same type (that is, the same `defstruct` name) and the values of all corresponding slots (slots having the same name) are `equalp`.

As part of the X3J13 discussion of this issue the following observations were made. Object equality is not a concept for which there is a uniquely determined correct algorithm. The appropriateness of an equality predicate can be judged only in the context of the needs of some particular program. Although these functions take any type of argument and their names sound very generic, `equal` and `equalp` are not appropriate for every application. Any decision to use or not use them should be determined by what they are documented to do rather than by any abstract characterization of their function. If neither `equal` nor `equalp` is found to be appropriate in a particular situation, programmers are encouraged to create another operator that is appropriate rather than blame `equal` or `equalp` for "doing the wrong thing."

Note that one consequence of the vote to change the rules of floating-point contagion (described in section 12.1) is to make `equalp` a true equivalence relation on numbers.

(equalp 'a 'b) ложь
(equalp 'a 'a) истина
(equalp 3 3) истина
(equalp 3 3.0) истина
(equalp 3.0 3.0) истина
(equalp #c(3 -4) #c(3 -4)) истина
(equalp #c(3 -4.0) #c(3 -4)) истина
(equalp (cons 'a 'b) (cons 'a 'c)) ложь
(equalp (cons 'a 'b) (cons 'a 'b)) истина
(equalp '(a . b) '(a . b)) истина
(progn (setq x (cons 'a 'b)) (equalp x x)) истина
(progn (setq x '(a . b)) (equalp x x)) истина
(equalp #\A #\A) истина

(equalp "Foo" "Foo") истина
(equalp "Foo" (copy-seq "Foo")) истина
(equalp "FOO" "foo") истина

6.4 Логические операторы

Common Lisp содержит три логических оператора для булевых значений: `and`, `or` и `not` (и, или, не, соответственно). `and` и `or` являются управляющими структурами, потому что их аргументы вычисляются в зависимости от условия. Функции `not` необходимо инвертировать её один аргумент, поэтому она может быть простой функцией.

[Функция] **not** *x*

`not` возвращает `t`, если *x* является `nil`, иначе возвращает `nil`. Таким образом она инвертирует аргумент как булево значение.

`null` то же, что и `not`, обе функции включены для ясности. По соглашению принято использовать `null`, когда надо проверить пустой ли список, и `not`, когда надо инвертировать булево значение.

[Макрос] **and** {form}*

`(and` *`form1 form2 ...`* `)` последовательно слева направо вычисляет формы. Если какая-либо форма *formN* вычислилась в `nil`, тогда немедленно возвращается значение `nil` без выполнения оставшихся форм. Если все формы кроме последней вычисляются в не-`nil` значение, `and` возвращает то, что вернула последняя форма. Таким образом, `and` может использоваться, как для логических операций, где `nil` обозначает ложь и не-`nil` значения истину, так и для условных выражений. Например:

```
(if (and (>= n 0)
         (< n (length a-simple-vector))
         (eq (elt a-simple-vector n) 'foo))
    (princ "Foo!"))
```

Выражение выше выводит `Foo!`, если `n`-ый элемент вектора `a-simple-vector` является символом `foo`, проверяя при этом вхождения `n` в границы вектора `a-simple-vector`. `elt` не будет вызвано с аргументом `n` выходящим за границы вектора, так как `and` гарантирует ленивую проверку аргументов слева направо.

Специальная форма Lisp'а `and` отличается тем, что в определённых случаях вычисляет не все аргументы.

Запись предыдущего примера

```
(and (>= n 0)
     (< n (length a-simple-vector))
     (eq (elt a-simple-vector n) 'foo)
     (princ "Foo!"))
```

будет выполнять ту же функцию. Разница в них только стилистическая. Некоторые программисты никогда не используют в форме `and` выражения с побочными эффектами, предпочитая для этих целей использовать `if` или `when`.

Из общего определения можно сделать дедуктивный вывод о том, что `(and` x`)` $\equiv x$. Также `(and)` выполняется в `t`, который тождественен этой операции.

Можно определить `and` в терминах `cond` таким образом:

```
(and x y z ... w) ≡ (cond ((not x) nil)
                          ((not y) nil)
                          ((not z) nil)
                          ...
                          (t w))
```

Смотрите `id` и `when`, которые иногда являются стилистически более удобными, чем `and` в целях ветвления. Если необходимо проверить истинность предиката для всех элементов списка или вектора (element 0 *and* element 1 *and* element 2 *and* ...), можно использовать функцию `every`.

[Макрос] **or** {form}*

`(or` *form1 form2 ...* `)` последовательно выполняет каждую форму слева направо. Если какая-либо непоследняя форма выполняется

в что-либо отличное от `nil`, `or` немедленно возвращает это не-`nil` значение без выполнения оставшихся форм. Если все формы кроме последней, вычисляются в `nil`, `or` возвращает то, что вернула последняя форма. Таким образом `or` может быть использована как для логических операций, в который `nil` обозначает ложь, и не-`nil` истину, так и для условного выполнения форм.

Специальная форма Lisp'а `or` отличается тем, что в определённых случаях вычисляет не все аргументы.

Из общего определения, можно сделать дедуктивный вывод о том, что `(or x)` $\equiv x$. Также, `(or)` выполняется в `nil`, который тождественен этой операции.

Можно определить `or` в терминах `cond` таким образом:

(or x y z ... w) $\equiv$ (cond (x) (y) (z) ... (`t` w))

Смотрите `id` и `unless`, которые иногда являются стилистически более удобными, чем `or` в целях ветвления. Если необходимо проверить истинность предиката для всех элементов списка или вектора (element 0 *or* element 1 *or* element 2 *or* ...), можно использовать функцию `some`.

Глава 7

Управляющие конструкции

Common Lisp предоставляет набор специальных структур для построения программ. Некоторые из них связаны со порядком выполнения (управляющие структуры), тогда как другие с управлением доступа к переменным (структурам окружения). Некоторые из этих возможностей реализованы как специальные формы; другие как макросы, которые в свою очередь разворачиваются в совокупность фрагментов программы, выраженных в терминах специальных форм или других макросов.

Вызов функции (применение функции) является основным методом создания Lisp программ. Операции записываются, как применение функции к её аргументам. Обычно Lisp программы пишутся, как большая совокупность маленьких функции, взаимодействующих с помощью вызовов одна другой, таким образом большие операции определяются в терминах меньших. Функции Lisp'а могут рекурсивно вызывать сами себя, как напрямую, так и косвенно.

Локально определённые функция (`flet`, `labels`) и макросы (`macrolet`) достаточно универсальны. Новая функциональность макросимволов позволяет использовать ещё больше синтаксической гибкости.

Несмотря на то, что язык Lisp более функциональный (applicative), чем императивный (statement-oriented), он предоставляет много операций, имеющих побочные эффекты, и следовательно требует конструкции для управления последовательностью вызовов с побочными эффектами. Конструкция `progn`, которая является приблизительным эквивалентом Algol'ному блоку **begin-end** с всеми его точками с

запятыми, последовательно выполняет некоторое количество форм, игнорируя все их значения, кроме последней. Много Lisp'овых управляющих конструкции неявно включают последовательное выполнение форм, в таком случае говорится, что это «неявный progn». Существуют также другие управляющие конструкции такие, как `prog1` и `prog2`.

Для циклов Common Lisp предоставляет, как общую функцию для итераций `do`, так и набор более специализированных функций для итераций или отображений (mapping) различных структур данных.

Common Lisp предоставляет простые одноветочные условные операторы `when` и `unless`, простой двуветочный условный оператор `if` и более общие многоветочные `cond` и `case`. Выбор одного из них для использования в какой-либо ситуации зависит от стиля и вкуса.

Предоставляются конструкции выполнения нелокальных выходов с различными правилами областей видимости: `block`, `return`, `return-from`, `catch` и `throw`.

Конструкции multiple-value предоставляют удобный способ для возврата более одного значения из функции, смотрите `values`.

7.1 Константы и переменные

Так как некоторые Lisp'овые объекты данных используются для отображения программ, можно всегда обозначить константный объект данных с помощью записи без приукрашательств формы данного объекта. Однако порождается двусмысленность: константный это объект или фрагмент кода. Эту двусмысленность разрешает специальная форма `quote`.

В Common Lisp'е присутствуют два вида переменных, а именно: обычные переменные и имена функций. Между этими типами есть несколько сходств, и в некоторых случаях для взаимодействия с ними используются похожие функции, например `boundp` и `fboundp`. Однако для в большинстве случаев два вида переменных используются для совсем разных целей: один указывает на функции, макросы и специальные формы, и другие на объекты данных.

X3J13 voted in March 1989 to introduce the concept of a *function-name*, which may be either a symbol or a two-element list whose first element is the symbol `setf` and whose second element is a symbol. The primary purpose of

this is to allow `setf` expander functions to be CLOS generic functions with user-defined methods. Many places in Common Lisp that used to require a symbol for a function name are changed to allow 2-lists as well; for example, `defun` is changed so that one may write `(defun (setf foo) ...)`, and the `function` special operator is changed to accept any function-name. See also `fdefinition`.

By convention, any function named `(setf` *f*`)` should return its first argument as its only value, in order to preserve the specification that `setf` returns its *newvalue*. See `setf`.

Implementations are free to extend the syntax of function-names to include lists beginning with additional symbols other than `setf` or `lambda`.

7.1.1 Ссылки на переменные

Значение обычной переменной может быть получено просто с помощью записи его имени как формы, которая будет выполнена. Будет ли данное имя распознано как имя специальной или лексической переменной зависит от наличия или отсутствия соответствующей декларации `special`. Смотрите главу 9.

Следующие функции и специальные формы позволяют ссылаться на значения констант и переменных.

[Специальный оператор] **quote** object

`(quote` *x*`)` возвращает *x*. *object* не выполняется и может быть любым объектом Lisp'a. Конструкция позволяет записать в программе любой объект, как константное значение. Например:

(setq a 43)
(list a (cons a 3)) ⇒ (43 (43 . 3))
(list (quote a) (quote (cons a 3)) ⇒ (a (cons a 3))

Так как `quote` форма так полезна, но записывать её трудоёмко, для неё определена стандартная аббревиатура: любая форма *f* с предшествующей одинарной кавычкой (') оборачивается формой `(quote )` для создания `(quote` *f*`)`. Например:

(setq x '(the magic quote hack))

обычно интерпретируется функцией `read`, как

```
(setq x (quote (the magic quote hack)))
```

Смотрите раздел 22.1.3.

It is an error to destructively modify any object that appears as a constant in executable code, whether within a `quote` special operator or as a self-evaluating form.

See section 24.1 for a discussion of how quoted constants are treated by the compiler.

Деструктивная модификация любого объекта, который представлен как константа в выполняемом коде с помощью специального оператора `quote` или как самовычисляемая форма, является ошибкой.

Смотрите раздел 24.1 для обсуждения того, как заквоченные константы обрабатываются компилятором.

X3J13 voted in March 1989 to clarify that `eval` and `compile` are not permitted either to copy or to coalesce ("collapse") constants (see `eq`) appearing in the code they process; the resulting program behavior must refer to objects that are `eql` to the corresponding objects in the source code. Moreover, the constraints introduced by the votes on issues and on what kinds of objects may appear as constants apply only to `compile-file` (see section 24.1).

[Специальный оператор] **function** fn

Значением `function` всегда является функциональной интерпретацией *fn*. *fn* интерпретируется как, если бы она была использована на позиции функции в форме вызова функции. В частности, если *fn* является символом, возвращается определение функции, связанное с этим символом, смотрите `symbol-function`. Если *fn* является лямбда-выражением, тогда возвращается «лексическое замыкание», это значит функция, которая при вызове выполняет тело лямбда-выражения таким образом, чтобы правила лексического контекста выполнялись правильно.

X3J13 voted in June 1988 to specify that the result of a `function` special operator is always of type `function`. This implies that a form `(function` *fn*`)` may be interpreted as `(the (function` *fn*`))`.

It is an error to use the `function` special operator on a symbol that does not denote a function in the lexical or global environment in which the

special operator appears. Specifically, it is an error to use the `function` special operator on a symbol that denotes a macro or special operator. Some implementations may choose not to signal this error for performance reasons, but implementations are forbidden to extend the semantics of `function` in this respect; that is, an implementation is not allowed to define the failure to signal an error to be a "useful" behavior.

Функция `function` принимает любое имя функции (символ или список, *car* элементы которого является `setf`—смотрите раздел 7.1) Также принимает принимает лямбда-выражение. Так можно записать `(function (setf cadr))` для ссылки на функцию раскрытия `setf` для `cadr`.

Например:

```
(defun adder (x) (function (lambda (y) (+ x y))))
```

Результат `(adder 3)` является функцией, которая добавляет `3` к её аргументу:

```
(setq add3 (adder 3))
(funcall add3 5) ⇒ 8
```

Это работает, потому что `function` создаёт замыкание над внутренним лямбда-выражением, которое может ссылаться на значение `3` переменной `x` даже после того, как выполнение вышло из функции `adder`.

Если посмотреть глубже, то лексическое замыкание обладает возможностью ссылаться на лексически видимые *связывание*, а не просто на значения. Рассмотрим такой код:

```
(defun two-funs (x)
  (list (function (lambda () x))
        (function (lambda (y) (setq x y)))))
(setq funs (two-funs 6))
(funcall (car funs)) ⇒ 6
(funcall (cadr funs) 43) ⇒ 43
(funcall (car funs)) ⇒ 43
```

Функция `two-funs` возвращает список двух функций, каждая из которых ссылается на *связывание* переменной `x`, созданной в момент входа в функцию `two-funs`, когда она была вызвана с аргументом `6`. Это связывание сначала имеет значение `6`, но `setq` может изменить связывание. Лексическое замыкание для первого лямбда-выражения не является «создаёт снимок» значения `6` для `x` при создании замыкания. Вторая функция может использоваться для изменения связывания (на `43` например), и это изменённое значение станет доступным в первой функции.

В ситуации, когда замыкание лямбда-выражения над одним и тем же множеством связываний может создаваться несколько раз, эти полученные разные замыкания могут быть равны или не равны `eq` в зависимости от реализации. Например:

```
(let ((x 5) (funs '()))
  (dotimes (j 10)
    (push #'(lambda (z)
              (if (null z) (setq x 0) (+ x z)))
          funs))
  funs)
```

Результат данного выражения является списком десяти замыканий. Каждое логически требует только связывания `x`. В любом случае это одно и то же связывание, но десять замыканий могут быть равны или не равны `eq` друг другу. С другой стороны, результат выражения

```
(let ((funs '()))
  (dotimes (j 10)
    (let ((x 5))
      (push (function (lambda (z)
                        (if (null z) (setq x 0) (+ x z))))
            funs)))
  funs)
```

также является списком из десяти замыканий. Однако в этом случае, но одна из пар замыканий не будет равна `eq`, потому что каждое

замыкание имеет своё связывание `x` отличное от другого. Связывания отличаются, так как в замыкании используется `setq`.

Вопрос различного поведения важен, поэтому рассмотрим следующее простое выражение:

```
(let ((funs '()))
  (dotimes (j 10)
    (let ((x 5))
      (push (function (lambda (z) (+ x z)))
            funs)))
  funs)
```

Результатом является десять замыканий, которые *могут* быть равны `eq` попарно. Однако, можно подумать что связывания `x` для каждого замыкания разные, так как создаются в цикле, но связывания не могут различаться, потому что их значения идентичны и неизменяемы (иммутабельны), в замыканиях отсутствует `setq` для `x`. Компилятор может в таких случаях оптимизировать выражение так:

```
(let ((funs '()))
  (dotimes (j 10)
    (push (function (lambda (z) (+ 5 z)))
          funs))
  funs)
```

после чего, в конце концов, замыкания точно могут быть равны. Общее правило такое, что реализация может в двух различных случаях выполнения формы `function` вернуть идентичные (`eq`) замыкания, если она может доказать, что два концептуально различающихся замыкания по факту ведут себя одинаково при одинаковых параметрах вызова. Это просто разрешается для оптимизации. Полностью корректная реализация может каждый раз при выполнении формы `function` возвращать новое замыкание не равное `eq` другим.

Часто компилятор может сделать вывод, что замыкание по факту не нуждается в замыкании над какими-либо связываниями переменных. Например, в фрагменте кода

```
(mapcar (function (lambda (x) (+ x 2))) y)
```

функция `(lambda (x) (+ x 2))` не содержит ссылок на какие-либо внешние сущности. В этом важном случае, одно и то же «замыкание» может быть использовано в качестве результата всех выполнений специальной формы `function`. Несомненно, данное значение может и не быть объектом замыкания. Оно может быть просто скомпилированной функцией, не содержащей информации об окружении. Данный пример просто является частным случаем предыдущего разговора и включён в качестве подсказки для разработчиков знакомых с предыдущими методами реализации Lisp'а. Различие между замыканиями и другими видами функций слегка размыто, Common Lisp не определяет отображения для замыканий и метода различия замыканий и простых функций. Все что имеет значение, это соблюдение правил лексической области видимости.

Так как форма `function` используются часто, но её запись длинная, для неё определена стандартная аббревиатура: любая форма *f* с предшествующими #' разворачивается в форму `(function f)`. Например,

```
(remove-if #'numberp '(1 a b 3))
```

обычно интерпретируется функцией `read` как

```
(remove-if (function numberp) '(1 a b 3))
```

Смотрите раздел 22.1.4.

[Функция] **symbol-value** *symbol*

`symbol-value` возвращает текущее значение динамической (специальной) переменной с именем *symbol*. Если символ не имеет значения, возникает ошибка. Смотрите `boundp` и `makunbound`. Следует отметить, что константные символы являются переменными, которые не могут быть изменены, таким образом `symbol-value` может использоваться для получения значения именованной константы. `symbol-value` от ключевого символа будет возвращать этот ключевой символ.

`symbol-value` не может получить доступ к значению лексической переменной.

В частности, эта функция полезна для реализации интерпретаторов для встраиваемых языков в Lisp'е. Соответствующая функция присваивания `set`. Кроме того, можно пользоваться конструкцией `setf` с `symbol-value`.

[Функция] **symbol-function** *symbol*

`symbol-function` возвращает текущее глобальное определение функции с именем *symbol*. В случае если символ не имеет определения функции сигнализируется ошибка, смотрите `fboundp`. Следует отметить что определение может быть функцией или объектом отображающим специальную форму или макрос. Однако, в последнем случае, попытка вызова объекта как функции будет является ошибкой. Лучше всего заранее проверить символ с помощью `macro-function` и `special-operator-p` и только затем вызвать функциональное значение, если оба предыдущих теста вернули ложь.

Эта функция полезна, в частности, для реализации интерпретаторов языков встроенных в Lisp.

`symbol-function` не может получить доступ к значению имени лексической функции, созданной с помощью `flet` или `labels`. Она может получать только глобальное функциональное значение.

Глобальное определение функции для некоторого символа может быть изменено с помощью `setf` и `symbol-function`. При использовании этой операции символ будет иметь *только* заданное определение в качестве своего глобального функционального значения. Любое предыдущее определение, было ли оно макросом или функцией, будет потеряно. Попытка переопределения специальной формы (смотрите таблицу 5.1) будет является ошибкой.

X3J13 voted in June 1988 to clarify the behavior of `symbol-function` in the light of the redefinition of the type `function`.

- It is permissible to call `symbol-function` on any symbol for which `fboundp` returns true. Note that `fboundp` must return true for a symbol naming a macro or a special operator.

- If `fboundp` returns true for a symbol but the symbol denotes a macro or special operator, then the value returned by `symbol-function` is not well-defined but `symbol-function` will not signal an error.

- When `symbol-function` is used with `setf` the new value must be of type `function`. It is an error to set the `symbol-function` of a symbol to a symbol, a list, or the value returned by `symbol-function` on the name of a macro or a special operator.

[Функция] **fdefinition** *function-name*

Функция похожа на `symbol-function` за исключением того, что её аргументы может быть любым именем функции (символ или списка, у которого *car* элемент равен `setf`—смотрите раздел 7.1). Функция возвращает текущее глобальное значение определения функции с именем *function-name*. Можно использовать `fdefinition` вместе с `setf` для изменения глобального определения функции связанной с переданным в параметре именем.

[Функция] **boundp** *symbol*

`boundp` является истиной, если динамическая (специальная) переменная с именем *symbol* имеет значение, иначе возвращает `nil`.

Смотрите также `set` и `makunbound`.

[Функция] **fboundp** *symbol*

`fboundp` является истиной, если символ имеет глобальное определение функции. Следует отметить, что `fboundp` является истиной, если символ указывает на специальную форму или макрос. `macro-function` и `special-operator-p` могут использоваться для проверки таких случаев.

Смотрите также `symbol-function` и `fmakunbound`.

Функция `fboundp` принимает любое имя функции (символ или список, *car* элементы которого является `setf`—смотрите раздел 7.1) Так можно записать `(fboundp '(setf cadr))` для определения существует ли функция `setf` для `cadr`.

[Функция] **special-operator-p** *symbol*

Функция `special-operator-p` принимает символ. Если символ указывает на специальную форму, тогда возвращается значение не-`nil`, иначе возвращается `nil`. Возвращённое не-`nil` значение является

функцией, которая может быть использована для интерпретации (вычисления) специальной формы. FIXME

Возможно также то, что *обе* функции `special-operator-p` и `macro-function` будут истинными для одного и того же символа. Это потому, что реализация может иметь любой макрос как специальную форму для скорости. С другой стороны, определение макроса должно быть доступно для использования программами, которые понимают только стандартные специальные формы, перечисленные в таблице 5.1. FIXME

7.1.2 Присваивание

Следующая функциональность позволяет изменять значение переменной (если быть точнее, значению соединённому с текущим связыванием переменной). Такое изменение отличается от создания нового связывания. Конструкции для создания новых связываний переменных описаны в разделе 7.5.

[Специальный оператор] **setq** {var form}*

Специальная форма (`setq` *var1 form1 var2 form2 ...*) является «конструкцией присваивания простых переменных» Lisp'а. Вычисляется первая форма *form1* и результат сохраняется в переменной *var1*, затем вычисляется *form2* и результат сохраняется в переменной *var2*, и так далее. Переменные, конечно же, представлены символами, и интерпретируются как ссылки к динамическим или статическим переменным в соответствии с обычными правилами. Таким образов `setq` может быть использована для присваивания как лексических, так и специальных переменных.

`setq` возвращает последнее присваиваемое значение, другими словами, результат вычисления последнего аргумента. В другом случае, форма (`setq`) является корректной и возвращает `nil`. В форме должно быть чётное количество форм аргументов. Например, в

```
(setq x (+ 3 2 1) y (cons x nil))
```

`x` устанавливается в `6`, `y` в `(6)`, и `setq` возвращает `(6)`. Следует отметить, что первое присваивание выполняется перед тем, как будет выполнено второе, тем самым каждое следующее присваивание может использовать значение предыдущих.

Смотрите также описание `setf`, «общая конструкция присваивания» Common Lisp'a, которая позволяет присваивать значения переменным, элементам массива, и другим местам.

Некоторые программисты выбирают путь отречения от `setq`, и всегда используют `setf`. Другие используют `setq` для простых переменных и `setf` для всех остальных.

Если любая *var* ссылается не на обычную переменную, а на связывание сделанное с помощью `symbol-macrolet`, тогда *var* обрабатывается как если `psetf` использовалось вместо `setq`.

[Макрос] **psetq** {var form}*

Форма `psetq` похожа на форму `setq` за исключением того, что выполняет присваивание параллельно. Сначала выполняются все формы, а затем переменные получают значения этих форм. Значение формы `psetq` `nil`. Например:

```
(setq a 1)
(setq b 2)
(psetq a b b a)
a ⇒ 2
b ⇒ 1
```

В этом примере, значения `a` и `b` меняются местами с помощью параллельного присваивания. (Если несколько переменных должны быть присвоены параллельно в рамках цикла, целесообразнее использовать конструкцию `do`.)

Смотрите также описание `psetf`, «общая конструкция параллельного присваивания» Common Lisp'a, которая позволяет присваивать переменным, элементам массива, и другим местам.

Если любая *var* ссылается не на обычную переменную, а на связывание сделанное с помощью `symbol-macrolet`, тогда *var* обрабатывается как если `psetf` использовалось вместо `psetq`.

[Функция] **set** *symbol value*

`set` позволяет изменить значение динамической (специальной) переменной. `set` устанавливает динамической переменной с именем *symbol* значение *value*.

Изменено будет только значение текущего динамического связывания. Если такого связывания нет, будет изменено наиболее глобальное значение. Например,

```
(set (if (eq a b) 'c 'd) 'foo)
```

установит значение `c` в `foo` или `do*` в `foo`, в зависимости от результата проверки `(eq a b)`.

`set` в качестве результата возвращает значение *value*.

`set` не может изменить значение локальной (лексически связанной) переменной. Обычно для изменения переменных (лексических или динамических) используется специальная форма `setq`. `set` полезна в частности для реализации интерпретаторов языков встроенных в Lisp. Смотрите также `progv`, конструкция, которая создаёт связывания, а не присваивания динамических переменных.

[Функция] **makunbound** *symbol*
[Функция] **fmakunbound** *symbol*

`makunbound` упраздняет связывание динамической (специальной) переменной заданной символом *symbol* (упраздняет значение). `fmakunbound` аналогично упраздняет связь символа с глобальным определением функции. Например:

```
(setq a 1)
a ⇒ 1
(makunbound 'a)
a ⇒ ошибка

(defun foo (x) (+ x 1))
(foo 4) ⇒ 5
(fmakunbound 'foo)
(foo 4) ⇒ ошибка
```

Обе функции возвращают символ *symbol* в качестве результата.

Функция `fmakunbound` принимает любое имя функции (символ или список, *car* элементы которого является `setf`—смотрите раздел 7.1) Так можно записать `(fmakunbound '(setf cadr))` для удаления функции `setf` для `cadr`.

7.2 Обобщённые переменные

В Lisp'е, переменная может запомнить одну часть данных, а точнее, один Lisp объект. Главные операции над переменной это получить её значение и задать ей другое значение. Их часто называют операциями *доступа* и *изменения*. Концепция переменных с именем в виде символа может быть обобщена до того, что любое место может сохранять в себе части данных вне зависимости от того, как данное место именуется. Примерами таких мест хранения являются *car* и *cdr* элементы cons-ячейки, элементы массива, и компоненты структуры.

Для каждого вида обобщённых переменных существуют две функции, которые реализуют операции *доступа* и *изменения*. Для переменных это имя переменной для доступа, а для изменения специальная форма `setq`. Функция `car` получает доступ к *car* элементу cons-ячейки, а функция `rplaca` изменяет этот элемент ячейки. Функция `symbol-value` получает динамическое значение переменной именованной некоторым символом, а функция `set` изменяет эту переменную.

Вместо того, чтобы думать о двух разных функциях, которые соответственно получают доступ и изменяют некоторое место хранения в зависимости от своих аргументов, мы может думать просто о вызове функции доступа с некоторыми аргументами, как о *имени* данного места хранения. Таким образом, просто `x` является именем места хранения (переменной), `(car x)` имя для *car* элементы для некоторой cons-ячейки (которая в свою очередь именуется символом `x`). Теперь вместо того, чтобы запоминать по две функции для каждого вида обобщённых переменных (например `rplaca` для `car`), мы адаптировали единый синтаксис для изменения некоторого места хранения с помощью макроса `setf`. Это аналогично способу, где мы используем специальную форму `setq` для преобразования имени переменной (которая является также формой для доступа к ней) в форму, которая изменяет переменную

FIXME. Эта универсальной отображения в следующей таблице.

Функция доступа	Функция изменения	Изменения с помощью `setf`
`x`	`(setq x datum)`	`(setf x datum)`
`(car x)`	`(rplaca x datum)`	`(setf (car x) datum)`
`(symbol-value x)`	`(set x datum)`	`(setf (symbol-value x) datum)`

`setf` это макрос, который анализирует форму доступа, и производит вызов соответствующей функции изменения.

С появление в Common Lisp'е `setf`, необходимость в `setq`, `rplaca` и `set` отпала. Они оставлены в Common Lisp из-за их исторической важности в Lisp'е. Однако, большинство других функций изменения (например `putprop`, функция изменения для `get`) были устранены из Common Lisp'а в расчёте на то, что везде на их месте будет использоваться `setf`.

[Макрос] **setf** {place newvalue}*

`(setf` ***`place newvalue`***`)` принимает форму *place*, которая при своём вычислении получает доступ к объекту в некотором месте хранения и «инвертирует» эту форму в соответствующую форму *изменения*. Таким образом вызов макроса `setf` разворачивается в форму изменения, которая сохраняет результат вычисления формы *newvalue* в место хранения, на которое ссылалась форма доступа.

Если пар *place-newvalue* указано более одной, эти пары обрабатываются последовательно. Таким образом:

(setf *place1 newvalue1*
place2 newvalue2)
...
placen newvaluen)

эквивалентно

(setf *place1 newvalue1*
place2 newvalue2)
...
placen newvaluen)

Следует отметить, что запись `(setf)` является корректной и возвращает `nil`.

Форма *place* может быть одной из следующих:

- Имя переменной (лексической и динамической).
- Формой вызова функции, у которой первый элемент принадлежит множеству указанному в следующей таблице:

`aref`	`car`	`svref`	
`nth`	`cdr`	`get`	
`elt`	`caar`	`getf`	`symbol-value`
`rest`	`cadr`	`gethash`	`symbol-function`
`first`	`cdar`	`documentation`	`symbol-plist`
`second`	`cddr`	`fill-pointer`	`macro-function`
`third`	`caaar`	`caaaar`	`cdaaar`
`fourth`	`caadr`	`caaadr`	`cdaadr`
`fifth`	`cadar`	`caadar`	`cdadar`
`sixth`	`caddr`	`caaddr`	`cdaddr`
`seventh`	`cdaar`	`cadaar`	`cddaar`
`eighth`	`cdadr`	`cadadr`	`cddadr`
`ninth`	`cddar`	`caddar`	`cdddar`
`tenth`	`cdddr`	`cadddr`	`cddddr`
`row-major-aref`	`compiler-macro-function`		

  Это правило применяется только тогда, когда имя функции ссылается глобальное определение функции, и не ссылается на локальное определение или макрос.

- Формой вызова функции, у которой первый элемент является именем функции-селектора созданной с помощью `defstruct`.

  Это правило применяется только тогда, когда имя функции ссылается глобальное определение функции, и не ссылается на локальное определение или макрос.

- A function call form whose first element is the name of any one of the following functions, provided that the new value

is of the specified type so that it can be used to replace the specified "location" (which is in each of these cases not truly a generalized variable):

Имя функции	Требуемый тип
`bit`	`bit`
`sbit`	`bit`
`subseq`	`sequence`

X3J13 voted in March 1989 to eliminate the type `string-char` and to redefine `string` to be the union of one or more specialized vector types, the types of whose elements are subtypes of the type `character`. In the preceding table, the type `string-char` should be replaced by some such phrase as "the element-type of the argument vector."

Это правило применяется только когда имя функции ссылается на глобальное, а не локальное определение функции.

In the case of `subseq`, the replacement value must be a sequence whose elements may be contained by the sequence argument to `subseq`. (Note that this is not so stringent as to require that the replacement value be a sequence of the same type as the sequence of which the subsequence is specified.) If the length of the replacement value does not equal the length of the subsequence to be replaced, then the shorter length determines the number of elements to be stored, as for the function `replace`.

- Форма вызова функции, первый элемент которой является именем одной из следующих функций при условии, что указанный аргумент этой функции в свою очередь является формой *place*. in this case the new *place* has stored back into it the result of applying the specified "update" function (which is in each of these cases not a true update function): FIXME

Имя функции	Аргумент являющийся *местом*	Функция изменения
`ldb`	second	`dpb`
`mask-field`	second	`deposit-field`

Это правило применяется только тогда, когда имя функции ссылается глобальное определение функции, и не ссылается на локальное определение или макрос.

- Форма декларации типа `the`, в таком случае декларация переносится на форму *newvalue*, и анализируется результирующая `setf` форма. Например:

  (setf (the integer (cadr x)) (+ y 3))

  будет обработана как

  (setf (cadr x) (the integer (+ y 3)))

- Вызов функции `apply`, в которой первый аргумент является функцией, которая может является *местом* для `setf`. That is, `(function` *`name`*`)`, where *name* is the name of a function, calls to which are recognized as places by `setf`. Suppose that the use of `setf` with `apply` looks like this:

  (setf (apply #'*name x1 x2 ... xn rest*) *x0*)

  The `setf` method for the function *name* must be such that

  (setf (*name z1 z2 ... zm*) *z0*)

  expands into a store form

  (*storefn* zi_1 zi_2 ... zi_k *zm*)

That is, it must expand into a function call such that all arguments but the last may be any permutation or subset of the new value *z0* and the arguments of the access form, but the *last* argument of the storing call must be the same as the last argument of the access call. See `define-setf-method` for more details on accessing and storing forms.

Given this, the `setf`-of-`apply` form shown above expands into

(apply #'*storefn* xi_1 xi_2 ... xi_k *rest*)

As an example, suppose that the variable `indexes` contains a list of subscripts for a multidimensional array *foo* whose rank is not known until run time. One may access the indicated element of the array by writing

(apply #'aref foo indexes)

and one may alter the value of the indicated element to that of `newvalue` by writing

(setf (apply #'aref foo indexes) newvalue)

Это правило применяется только тогда, когда имя функции ссылается глобальное определение функции, и не ссылается на локальное определение или макрос.

- Макровызов, в случае чего `setf` разворачивает макровызов и затем анализирует полученную форму.

  Этот шаг используется `macroexpand-1`, но не `macroexpand`. Это позволяет применить какое-либо из предшествующих правил.

- Любая форма, для которой было сделано определение с помощью `defsetf` или `define-setf-method`.

  Это правило применяется только тогда, когда имя функции ссылается глобальное определение функции, и не ссылается на локальное определение или макрос.

X3J13 voted in March 1989 to add one more rule to the preceding list, coming after all those listed above:

- Any other list whose first element is a symbol (call it *f*). In this case, the call to `setf` expands into a call to the function named by the list `(setf` *f*`)` (see section 7.1). The first argument is the new value and the remaining arguments are the values of the remaining elements of *place*. This expansion occurs regardless of whether either *f* or `(setf` *f*`)` is defined as a function locally, globally, or not at all. For example,

  (setf (*f arg1 arg2 ...*) *newvalue*)

  expands into a form with the same effect and value as

```
(let ((#:temp1 arg1)     ;Force correct order of evaluation
      (#:temp2 arg2)
      ...
      (#:temp0 newvalue))
  (funcall (function (setf f))
           #:temp0
           #:temp1
           #:temp2 ...))
```

  By convention, any function named `(setf` *f*`)` should return its first argument as its only value, in order to preserve the specification that `setf` returns its *newvalue*.

X3J13 voted in March 1989 to add this case as well:

- A variable reference that refers to a symbol macro definition made by `symbol-macrolet`, in which case `setf` expands the reference and then analyzes the resulting form.

`setf` тщательно сохраняет обычный порядок выполнения подформ слева направо. С другой стороны, точное раскрытие для какой-нибудь частной формы не гарантируется и может зависеть от реализации. Все, что гарантируется, это раскрытие `setf` формы в некоторую функцию

изменения, используемую данной реализацией, и выполнение подформ слева направо.

Конечным результатом вычисления формы `setf` является значение *newvalue*. Таким образом `(setf (car x) y)` раскрывается не прямо в `(rplaca x y)`, а в что-то вроде

```
(let ((G1 x) (G2 y)) (rplaca G1 G2) G2)
```

точное раскрытие зависит от реализации.

Пользователь может определить новое раскрытие для `setf` используя `defsetf`.

X3J13 voted in June 1989 to extend the specification of `setf` to allow a *place* whose `setf` method has more than one store variable (see `define-setf-method`). In such a case as many values are accepted from the *newvalue* form as there are store variables; extra values are ignored and missing values default to `nil`, as is usual in situations involving multiple values.

A proposal was submitted to X3J13 in September 1989 to add a `setf` method for `values` so that one could in fact write, for example,

```
(setf (values quotient remainder)
      (truncate linewidth tabstop))
```

but unless this proposal is accepted users will have to define a `setf` method for `values` themselves (not a difficult task).

[Макрос] **psetf** {place newvalue}*

`psetf` похожа на `setf` за исключением того, что если указано более одной пары *place-newvalue* , то присваивание местам новых значений происходит параллельно. Если говорить точнее, то все подформы, которые должны быть вычислены, вычисляются слева направо. После выполнения всех вычислений, выполняются все присваивания в неопределённом порядке. (Неопределённый порядок влияет на поведение в случае, если более одной формы *place* ссылаются на одно и то же место.) `psetf` всегда возвращает `nil`.

X3J13 voted in June 1989 to extend the specification of `psetf` to allow a *place* whose `setf` method has more than one store variable (see

`define-setf-method`). In such a case as many values are accepted from the *newvalue* form as there are store variables; extra values are ignored and missing values default to `nil`, as is usual in situations involving multiple values.

[Макрос] **shiftf** {place}+ newvalue

Каждая форма *place* может быть любой обобщённой переменной, как для `setf`. В форме (`shiftf` *place1 place2 ... placen newvalue*), вычисляются и сохраняются значения с *place1* по *placen* и вычисляется *newvalue*, как значение с номером $n + 1$. Значения с 2-го по $n + 1$ сохраняются в интервале с *place1* по *placen* и возвращается 1-ое значение (оригинальное значение *place1*). Механизм работает как сдвиг регистров. *newvalue* сдвигается с правой стороны, все значения сдвигаются влево на одну позицию, и возвращается сдвигаемое самое левое значение *place1*. Например:

(setq x (list 'a 'b 'c)) ⇒ (a b c)

(shiftf (cadr x) 'z) ⇒ b
  and now x ⇒ (a z c)

(shiftf (cadr x) (cddr x) 'q) ⇒ z
  and now x ⇒ (a (c) . q)

Эффект от (`shiftf` *place1 place2 ... placen newvalue*) эквивалентен

(let ((*var1 place1*)
      (*var2 place2*)
      ...
      (*varn placen*))
  (setf *place1 var2*)
  (setf *place2 var3*)
  ...
  (setf *placen newvalue*)
  var1)

за исключением того, что последний вариант выполняет все подформы для каждого *place* дважды, тогда как `shiftf` выполняет только единожды. Например:

(setq n 0)
(setq x '(a b c d))
(shiftf (nth (setq n (+ n 1)) x) 'z) ⇒ b
  теперь x ⇒ (a z c d)

but

(setq n 0)
(setq x '(a b c d))
(prog1 (nth (setq n (+ n 1)) x)
       (setf (nth (setq n (+ n 1)) x) 'z)) ⇒ b
  и теперь x ⇒ (a b z d)

Более того, для заданных форм *place* `shiftf` может быть более производительной, чем версия с `prog1`.

X3J13 voted in June 1989 to extend the specification of `shiftf` to allow a *place* whose `setf` method has more than one store variable (see `define-setf-method`). In such a case as many values are accepted from the *newvalue* form as there are store variables; extra values are ignored and missing values default to `nil`, as is usual in situations involving multiple values.

Обоснование: `shiftf` and `rotatef` have been included in Common Lisp as generalizations of two-argument versions formerly called `swapf` and `exchf`. The two-argument versions have been found to be very useful, but the names were easily confused. The generalization to many argument forms and the change of names were both inspired by the work of Suzuki [47], which indicates that use of these primitives can make certain complex pointer-manipulation programs clearer and easier to prove correct.

[Макрос] **rotatef** {place}*

Каждая *place* может быть обобщённой переменной, как для `setf`. В форме (`rotatef` *place1 place2 ... placen*), вычисляются и сохраняются значения с *place1* по *placen*. Механизм действует как круговой сдвиг регистров влево, и значение *place1* сдвигается в конец

на *placen*. Следует отметить, что `(rotatef` *`place1 place2`*`)` меняет значения между *place1* и *place2*.

Эффект от использования `(rotatef` *`place1 place2 ... placen`*`)` эквивалентен

(psetf *place1 place2*
place2 place3
...
placen place1)

за исключением того, что в последнем вычисление форм происходит дважды, тогда как `rotatef` выполняет только единожды. Более того, для заданных форм *place* `rotatef` может быть более производительной, чем версия с `prog1`.

`rotatef` всегда возвращает `nil`.

X3J13 voted in June 1989 to extend the specification of `rotatef` to allow a *place* whose `setf` method has more than one store variable (see `define-setf-method`). In such a case as many values are accepted from the *newvalue* form as there are store variables; extra values are ignored and missing values default to `nil`, as is usual in situations involving multiple values.

Другие макросы, которые управляют обобщёнными переменными, включают `getf`, `remf`, `incf`, `decf`, `push`, `pop`, `assert`, `ctypecase` и `ccase`.

Макросы, которые управляют обобщёнными переменными, должны гарантировать «явную» семантику: подформы обобщённых переменных вычисляются точно столько раз, сколько они встречаются в выражении, и в том же порядке, в котором встречаются.

В ссылках на обобщённые переменные, как в `shiftf`, `incf`, `push` и `setf` или `ldb`, обобщённые переменные считываются и записываются в одну и ту же ссылку. Сохранение порядка выполнения исходной программы и количества выполнений чрезвычайно важно.

As an example of these semantic rules, in the generalized-variable reference `(setf` *`reference value`*`)` the *value* form must be evaluated *after* all the subforms of the reference because the *value* form appears to the right of them.

The expansion of these macros must consist of code that follows these rules or has the same effect as such code. This is accomplished by introducing temporary variables bound to the subforms of the reference. As an

optimization in the implementation, temporary variables may be eliminated whenever it can be proved that removing them has no effect on the semantics of the program. For example, a constant need never be saved in a temporary variable. A variable, or for that matter any form that does not have side effects, need not be saved in a temporary variable if it can be proved that its value will not change within the scope of the generalized-variable reference.

Common Lisp provides built-in facilities to take care of these semantic complications and optimizations. Since the required semantics can be guaranteed by these facilities, the user does not have to worry about writing correct code for them, especially in complex cases. Even experts can become confused and make mistakes while writing this sort of code.

X3J13 voted in March 1988 to clarify the preceding discussion about the order of evaluation of subforms in calls to `setf` and related macros. The general intent is clear: evaluation proceeds from left to right whenever possible. However, the left-to-right rule does not remove the obligation on writers of macros and `define-setf-method` to work to ensure left-to-right order of evaluation.

Let it be emphasized that, in the following discussion, a *form* is something whose syntactic use is such that it will be evaluated. A *subform* means a form that is nested inside another form, not merely any Lisp object nested inside a form regardless of syntactic context.

The evaluation ordering of subforms within a generalized variable reference is determined by the order specified by the second value returned by `get-setf-method`. For all predefined generalized variable references (`getf`, `ldb`), this order of evaluation is exactly left-to-right. When a generalized variable reference is derived from a macro expansion, this rule is applied *after* the macro is expanded to find the appropriate generalized variable reference.

This is intended to make it clear that if the user writes a `defmacro` or `define-setf-method` macro that doesn't preserve left-to-right evaluation order, the order specified in the user's code holds. For example, given

(defmacro wrong-order (x y) ‘(getf ,y ,x))

then

(push *value* (wrong-order *place1 place2*))

will evaluate *place2* first and then *place1* because that is the order they are evaluated in the macro expansion.

For the macros that manipulate generalized variables (`push`, `pushnew`, `getf`, `remf`, `incf`, `decf`, `shiftf`, `rotatef`, `psetf`, `setf`, `pop`, and those defined with `define-modify-macro`) the subforms of the macro call are evaluated exactly once in left-to-right order, with the subforms of the generalized variable references evaluated in the order specified above.

Each of `push`, `pushnew`, `getf`, `remf`, `incf`, `decf`, `shiftf`, `rotatef`, `psetf`, and `pop` evaluates all subforms before modifying any of the generalized variable locations. Moreover, `setf` itself, in the case when a call on it has more than two arguments, performs its operation on each pair in sequence. That is, in

(setf *place1 value1 place2 value2* ...)

the subforms of *place1* and *value1* are evaluated, the location specified by *place1* is modified to contain the value returned by *value1*, and then the rest of the `setf` form is processed in a like manner.

For the macros `check-type`, `ctypecase`, and `ccase`, subforms of the generalized variable reference are evaluated once per test of a generalized variable, but they may be evaluated again if the type check fails (in the case of `check-type`) or if none of the cases holds (in `ctypecase` or `ccase`).

For the macro `assert`, the order of evaluation of the generalized variable references is not specified.

Another reason for building in these functions is that the appropriate optimizations will differ from implementation to implementation. In some implementations most of the optimization is performed by the compiler, while in others a simpler compiler is used and most of the optimization is performed in the macros. The cost of binding a temporary variable relative to the cost of other Lisp operations may differ greatly between one implementation and another, and some implementations may find it best never to remove temporary variables except in the simplest cases.

A good example of the issues involved can be seen in the following generalized-variable reference:

(incf (ldb byte-field variable))

This ought to expand into something like

```
(setq variable
      (dpb (1+ (ldb byte-field variable))
           byte-field
           variable))
```

In this expansion example we have ignored the further complexity of returning the correct value, which is the incremented byte, not the new value of `variable`. Note that the variable `byte-field` is evaluated twice, and the variable `variable` is referred to three times: once as the location in which to store a value, and twice during the computation of that value.

Now consider this expression:

```
(incf (ldb (aref byte-fields (incf i))
           (aref (determine-words-array) i)))
```

It ought to expand into something like this:

```
(let ((temp1 (aref byte-fields (incf i)))
      (temp2 (determine-words-array)))
  (setf (aref temp2 i)
        (dpb (1+ (ldb temp1 (aref temp2 i)))
             temp1
             (aref temp2 i))))
```

Again we have ignored the complexity of returning the correct value. What is important here is that the expressions `(incf i)` and `(determine-words-array)` must not be duplicated because each may have a side effect or be affected by side effects.

X3J13 voted in January 1989 to specify more precisely the order of evaluation of subforms when `setf` is used with an access function that itself takes a *place* as an argument, for example, `ldb`, `mask-field`, and `getf`. (The vote also discussed the function `char-bit`, but another vote removed that function from the language.) The `setf` methods for such accessors produce expansions that effectively require explicit calls to `get-setf-method`.

The code produced as the macro expansion of a `setf` form that itself admits a generalized variable as an argument must essentially do the following major steps:

- It evaluates the value-producing subforms, in left-to-right order, and binds the temporary variables to them; this is called *binding the temporaries*.
- It reads the value from the generalized variable, using the supplied accessing form, to get the old value; this is called *doing the access*. Note that this is done after all the evaluations of the preceding step, including any side effects they may have.
- It binds the store variable to a new value, and then installs this new value into the generalized variable using the supplied storing form; this is called *doing the store*.

Doing the access for a generalized variable reference is not part of the series of evaluations that must be done in left-to-right order.

The place-specifier forms `ldb`, `mask-field`, and `getf` admit (other) *place* specifiers as arguments. During the `setf` expansion of these forms, it is necessary to call `get-setf-method` to determine how the inner, nested generalized variable must be treated.

In a form such as

(setf (ldb *byte-spec place-form*) *newvalue-form*)

the place referred to by the *place-form* must always be both accessed and updated; note that the update is to the generalized variable specified by *place-form*, not to any object of type `integer`.

Thus this call to `setf` should generate code to do the following:

- Evaluate *byte-spec* and bind into a temporary
- Bind the temporaries for *place-form*
- Evaluate *newvalue-form* and bind into the store variable
- Do the access to *place-form*

- Do the store into *place-form* with the given bit-field of the accessed integer replaced with the value in the store variable

If the evaluation of *newvalue-form* alters what is found in the given *place*—such as setting a different bit-field of the integer—then the change of the bit-field denoted by *byte-spec* will be to that altered integer, because the access step must be done after the *newvalue-form* evaluation. Nevertheless, the evaluations required for binding the temporaries are done before the evaluation of the *newvalue-form*, thereby preserving the required left-to-right evaluation order.

The treatment of `mask-field` is similar to that of `ldb`.

In a form such as:

(setf (getf *place-form ind-form*) *newvalue-form*)

the place referred to by the *place-form* must always be both accessed and updated; note that the update is to the generalized variable specified by *place-form*, not necessarily to the particular list which is the property list in question.

Thus this call to `setf` should generate code to do the following:

- Bind the temporaries for *place-form*
- Evaluate *ind-form* and bind into a temporary
- Evaluate the *newvalue-form* and bind into the store variable
- Do the access to *place-form*
- Do the store into *place-form* with a possibly new property list obtained by combining the results of the evaluations and the access

If the evaluation of *newvalue-form* alters what is found in the given *place*—such as setting a different named property in the list—then the change of the property denoted by *ind-form* will be to that altered list, because the access step is done after the *newvalue-form* evaluation. Nevertheless, the evaluations required for binding the temporaries are done before the evaluation of the *newvalue-form*, thereby preserving the required left-to-right evaluation order.

Note that the phrase "possibly new property list" treats the implementation of property lists as a "black box"; it can mean that the former property list is somehow destructively re-used, or it can mean partial or full copying of it. A side effect may or may not occur; therefore `setf` must proceed as if the resultant property list were a different copy needing to be stored back into the generalized variable.

The Common Lisp facilities provided to deal with these semantic issues include:

- Built-in macros such as `setf` and `push` that follow the semantic rules.
- The `define-modify-macro` macro, which allows new generalized-variable manipulating macros (of a certain restricted kind) to be defined easily. It takes care of the semantic rules automatically.
- The `defsetf` macro, which allows new types of generalized-variable references to be defined easily. It takes care of the semantic rules automatically.
- The `define-setf-method` macro and the `get-setf-method` function, which provide access to the internal mechanisms when it is necessary to define a complicated new type of generalized-variable reference or generalized-variable-manipulating macro.

Also important are the changes that allow lexical environments to be used in appropriate ways in `setf` methods.

[Макрос] **define-modify-macro** name lambda-list function [doc-string]

This macro defines a read-modify-write macro named *name*. An example of such a macro is `incf`. The first subform of the macro will be a generalized-variable reference. The *function* is literally the function to apply to the old contents of the generalized-variable to get the new contents; it is not evaluated. *lambda-list* describes the remaining arguments for the *function*; these arguments come from the remaining subforms of the macro after the generalized-variable reference. *lambda-list* may contain `&optional` and `&rest` markers. (The `&key` marker is not permitted here; `&rest` suffices for the purposes of `define-modify-macro`.) *doc-string* is documentation for the macro *name* being defined.

The expansion of a `define-modify-macro` is equivalent to the following, except that it generates code that follows the semantic rules outlined above.

```
(defmacro name (reference . lambda-list)
  doc-string
  `(setf ,reference
         (function ,reference ,arg1 ,arg2 ...)))
```

where *arg1*, *arg2*, ..., are the parameters appearing in *lambda-list*; appropriate provision is made for a `&rest` parameter.

As an example, `incf` could have been defined by:

```
(define-modify-macro incf (&optional (delta 1)) +)
```

An example of a possibly useful macro not predefined in Common Lisp is

```
(define-modify-macro unionf (other-set &rest keywords) union)
```

X3J13 voted in March 1988 to specify that `define-modify-macro` creates macros that take `&environment` arguments and perform the equivalent of correctly passing such lexical environments to `get-setf-method` in order to correctly maintain lexical references.

[Макрос] **defsetf** access-fn {update-fn [doc-string] | lambda-list (store-variable) [[{declaration}* | d

This defines how to `setf` a generalized-variable reference of the form `(access-fn ...)`. The value of a generalized-variable reference can always be obtained simply by evaluating it, so *access-fn* should be the name of a function or a macro.

The user of `defsetf` provides a description of how to store into the generalized-variable reference and return the value that was stored (because `setf` is defined to return this value). The implementation of `defsetf` takes care of ensuring that subforms of the reference are evaluated exactly once and in the proper left-to-right order. In order to do this, `defsetf` requires that *access-fn* be a function or a macro that evaluates its arguments, behaving like a function. Furthermore, a `setf` of a call on *access-fn* will also evaluate all of *access-fn*'s arguments; it cannot treat any of them specially. This means that `defsetf` cannot be used to describe how to store into a generalized variable that is a byte, such as `(ldb field reference)`. To handle situations that

do not fit the restrictions imposed by `defsetf`, use `define-setf-method`, which gives the user additional control at the cost of increased complexity.

A `defsetf` declaration may take one of two forms. The simple form is

(defsetf *access-fn update-fn* [doc-string])

The *update-fn* must name a function (or macro) that takes one more argument than *access-fn* takes. When `setf` is given a *place* that is a call on *access-fn*, it expands into a call on *update-fn* that is given all the arguments to *access-fn* and also, as its last argument, the new value (which must be returned by *update-fn* as its value). For example, the effect of

(defsetf symbol-value set)

is built into the Common Lisp system. This causes the expansion

(setf (symbol-value foo) fu) → (set foo fu)

for example. Note that

(defsetf car rplaca)

would be incorrect because `rplaca` does not return its last argument.

The complex form of `defsetf` looks like

(defsetf *access-fn lambda-list* (*store-variable*) . *body*)

and resembles `defmacro`. The *body* must compute the expansion of a `setf` of a call on *access-fn*.

The *lambda-list* describes the arguments of *access-fn*. `&optional`, `&rest`, and `&key` markers are permitted in *lambda-list*. Optional arguments may have defaults and "supplied-p" flags. The *store-variable* describes the value to be stored into the generalized-variable reference.

Обоснование: The *store-variable* is enclosed in parentheses to provide for an extension to multiple store variables that would receive multiple values from the second subform of `setf`. The rules given below for coding `setf` methods discuss

the proper handling of multiple store variables to allow for the possibility that this extension may be incorporated into Common Lisp in the future.

The *body* forms can be written as if the variables in the *lambda-list* were bound to subforms of the call on *access-fn* and the *store-variable* were bound to the second subform of `setf`. However, this is not actually the case. During the evaluation of the *body* forms, these variables are bound to names of temporary variables, generated as if by `gensym` or `gentemp`, that will be bound by the expansion of `setf` to the values of those subforms. This binding permits the *body* forms to be written without regard for order-of-evaluation issues. `defsetf` arranges for the temporary variables to be optimized out of the final result in cases where that is possible. In other words, an attempt is made by `defsetf` to generate the best code possible in a particular implementation.

Note that the code generated by the *body* forms must include provision for returning the correct value (the value of *store-variable*). This is handled by the *body* forms rather than by `defsetf` because in many cases this value can be returned at no extra cost, by calling a function that simultaneously stores into the generalized variable and returns the correct value.

An example of the use of the complex form of `defsetf`:

```
(defsetf subseq (sequence start &optional end) (new-sequence)
  `(progn (replace ,sequence ,new-sequence
                   :start1 ,start :end1 ,end)
          ,new-sequence))
```

X3J13 voted in March 1988 to specify that the body of the expander function defined by the complex form of `defsetf` is implicitly enclosed in a `block` construct whose name is the same as the *name* of the *access-fn*. Therefore `return-from` may be used to exit from the function.

X3J13 voted in March 1989 to clarify that, while defining forms normally appear at top level, it is meaningful to place them in non-top-level contexts; the complex form of `defsetf` must define the expander function within the enclosing lexical environment, not within the global environment.

The underlying theory by which `setf` and related macros arrange to conform to the semantic rules given above is that from any generalized-variable reference one may derive its "`setf` method," which describes how to store into that reference and which subforms of it are evaluated.

Given knowledge of the subforms of the reference, it is possible to avoid evaluating them multiple times or in the wrong order. A `setf` method for a given access form can be expressed as five values:

- A list of *temporary variables*
- A list of *value forms* (subforms of the given form) to whose values the temporary variables are to be bound
- A second list of temporary variables, called *store variables*
- A *storing form*
- An *accessing form*

The temporary variables will be bound to the values of the value forms as if by `let*`; that is, the value forms will be evaluated in the order given and may refer to the values of earlier value forms by using the corresponding variables.

The store variables are to be bound to the values of the *newvalue* form, that is, the values to be stored into the generalized variable. In almost all cases only a single value is to be stored, and there is only one store variable.

The storing form and the accessing form may contain references to the temporary variables (and also, in the case of the storing form, to the store variables). The accessing form returns the value of the generalized variable. The storing form modifies the value of the generalized variable and guarantees to return the values of the store variables as its values; these are the correct values for `setf` to return. (Again, in most cases there is a single store variable and thus a single value to be returned.) The value returned by the accessing form is, of course, affected by execution of the storing form, but either of these forms may be evaluated any number of times and therefore should be free of side effects (other than the storing action of the storing form).

The temporary variables and the store variables are generated names, as if by `gensym` or `gentemp`, so that there is never any problem of name clashes among them, or between them and other variables in the program. This is necessary to make the special operators that do more than one `setf` in parallel work properly; these are `psetf`, `shiftf`, and `rotatef`. Computation of the `setf` method must always create new variable names; it may not return the same ones every time.

Some examples of `setf` methods for particular forms:

- For a variable `x`:

  ()
  ()
  (g0001)
  (setq x g0001)
  x

- For `(car` *`exp`*`)`:

  (g0002)
  (*exp*)
  (g0003)
  (progn (rplaca g0002 g0003) g0003)
  (car g0002)

- For `(subseq` *`seq s e`*`)`:

  (g0004 g0005 g0006)
  (*seq s e*)
  (g0007)
  (progn (replace g0004 g0007 :start1 g0005 :end1 g0006)
  g0007)
  (subseq g0004 g0005 g0006)

[Макрос] **define-setf-method** access-fn lambda-list [[{declaration}* | doc-string]] {form}*

This defines how to `setf` a generalized-variable reference that is of the form `(`*`access-fn`*`...)`. The value of a generalized-variable reference can always be obtained simply by evaluating it, so *access-fn* should be the name of a function or a macro.

The *lambda-list* describes the subforms of the generalized-variable reference, as with `defmacro`. The result of evaluating the *forms* in the body must be five values representing the `setf` method, as described above. Note

that `define-setf-method` differs from the complex form of `defsetf` in that while the body is being executed the variables in *lambda-list* are bound to parts of the generalized-variable reference, not to temporary variables that will be bound to the values of such parts. In addition, `define-setf-method` does not have `defsetf`'s restriction that *access-fn* must be a function or a function-like macro; an arbitrary `defmacro` destructuring pattern is permitted in *lambda-list*.

By definition there are no good small examples of `define-setf-method` because the easy cases can all be handled by `defsetf`. A typical use is to define the `setf` method for `ldb`:

```
;;; SETF method for the form (LDB bytespec int).
;;; Recall that the int form must itself be suitable for SETF.
(define-setf-method ldb (bytespec int)
  (multiple-value-bind (temps vals stores
                        store-form access-form)
      (get-setf-method int)          ;Get SETF method for int
    (let ((btemp (gensym))           ;Temp var for byte specifier
          (store (gensym))           ;Temp var for byte to store
          (stemp (first stores)))    ;Temp var for int to store
      ;; Return the SETF method for LDB as five values.
      (values (cons btemp temps)     ;Temporary variables
              (cons bytespec vals)   ;Value forms
              (list store)           ;Store variables
              `(let ((,stemp (dpb ,store ,btemp ,access-form)))
                 ,store-form
                 ,store)             ;Storing form
              `(ldb ,btemp ,access-form)     ;Accessing form
              ))))
```

X3J13 voted in March 1988 to specify that the `&environment` lambda-list keyword may appear in the *lambda-list* in the same manner as for `defmacro` in order to obtain the lexical environment of the call to the `setf` macro. The preceding example should be modified to take advantage of this new feature. The `setf` method must accept an `&environment` parameter, which will receive the lexical environment of the call to `setf`; this environment must then be given to `get-setf-method` in order that it may correctly use

any locally bound `setf` method that might be applicable to the *place* form that appears as the second argument to `ldb` in the call to `setf`.

```
;;; SETF method for the form (LDB bytespec int).
;;; Recall that the int form must itself be suitable for SETF.
;;; Note the use of an &environment parameter to receive the
;;; lexical environment of the call for use with GET-SETF-METHOD.
(define-setf-method ldb (bytespec int &environment env)
  (multiple-value-bind (temps vals stores
                        store-form access-form)
      (get-setf-method int env)     ;Get SETF method for int
    (let ((btemp (gensym))          ;Temp var for byte specifier
          (store (gensym))          ;Temp var for byte to store
          (stemp (first stores)))   ;Temp var for int to store
      ;; Return the SETF method for LDB as five values.
      (values (cons btemp temps)    ;Temporary variables
              (cons bytespec vals)  ;Value forms
              (list store)          ;Store variables
              `(let ((,stemp (dpb ,store ,btemp ,access-form)))
                 ,store-form
                 ,store)                    ;Storing form
              `(ldb ,btemp ,access-form)    ;Accessing form
              ))))
```

X3J13 voted in March 1988 to specify that the body of the expander function defined by `define-setf-method` is implicitly enclosed in a `block` construct whose name is the same as the *name* of the *access-fn*. Therefore `return-from` may be used to exit from the function.

X3J13 voted in March 1989 to clarify that, while defining forms normally appear at top level, it is meaningful to place them in non-top-level contexts; `define-setf-method` must define the expander function within the enclosing lexical environment, not within the global environment.

X3J13 voted in March 1988 to add an optional environment argument to `get-setf-method`. The revised definition and example are as follows.

[Function] **get-setf-method** *form* **&optional** *env*

`get-setf-method` returns five values constituting the `setf` method for

form. The *form* must be a generalized-variable reference. The *env* must be an environment of the sort obtained through the `&environment` lambda-list keyword; if *env* is `nil` or omitted, the null lexical environment is assumed. `get-setf-method` takes care of error checking and macro expansion and guarantees to return exactly one store variable.

As an example, an extremely simplified version of `setf`, allowing no more and no fewer than two subforms, containing no optimization to remove unnecessary variables, and not allowing storing of multiple values, could be defined by:

```
(defmacro setf (reference value &environment env)
  (multiple-value-bind (vars vals stores store-form access-form)
      (get-setf-method reference env)     ;Note use of environment
    (declare (ignore access-form))
    `(let* ,(mapcar #'list
                    (append vars stores)
                    (append vals (list value)))
       ,store-form)))
```

X3J13 voted in March 1988 to add an optional environment argument to `get-setf-method`. The revised definition is as follows.

[Function] **get-setf-method-multiple-value** *form* **&optional** *env*

`get-setf-method-multiple-value` returns five values constituting the `setf` method for *form*. The *form* must be a generalized-variable reference. The *env* must be an environment of the sort obtained through the `&environment` lambda-list keyword; if *env* is `nil` or omitted, the null lexical environment is assumed.

This is the same as `get-setf-method` except that it does not check the number of store variables; use this in cases that allow storing multiple values into a generalized variable. There are no such cases in standard Common Lisp, but this function is provided to allow for possible extensions.

X3J13 voted in March 1988 to clarify that a `setf` method for a functional name is applicable only when the global binding of that name is lexically visible. If such a name has a local binding introduced by `flet`, `labels`, or `macrolet`, then global definitions of `setf` methods for that name do not apply and are not visible. All of the standard Common Lisp macros that

modify a `setf` *place* (for example, `incf`, `decf`, `pop`, and `rotatef`) obey this convention.

7.3 Вызов функции

Наиболее примитивная форма вызова функции в Lisp'е, конечно, не имеет имени. Любой список, который не интерпретируется как макровызов или вызов специальной формы, рассматривается как вызов функции. Другие конструкции предназначены для менее распространённых, но тем не менее полезных ситуаций.

[Функция] **apply** *function arg* **&rest** *more-args*

Функция применяет функцию *function* к списку аргументов.

X3J13 voted in June 1988 to allow the *function* to be only of type `symbol` or `function`; a lambda-expression is no longer acceptable as a functional argument. One must use the `function` special operator or the abbreviation `#'` before a lambda-expression that appears as an explicit argument form.

Все аргументы, кроме *function* передаются в применяемую функцию. Если последний аргумент список, то он присоединяется в конец к списку первых аргументов. Например:

(setq f '+) (apply f '(1 2)) ⇒ 3
(setq f #'-) (apply f '(1 2)) ⇒ -1
(apply #'max 3 5 '(2 7 3)) ⇒ 7
(apply 'cons '((+ 2 3) 4)) ⇒
((+ 2 3) . 4) *not* (5 . 4)
(apply #'+ '()) ⇒ 0

Следует отметить, что если функция принимает именованные аргументы, ключевые символы должны быть перечислены в списке аргументов так же, как и обычные значения:

(apply #'(lambda (`&key` a b) (list a b)) '(:b 3)) ⇒ (`nil` 3)

Это бывает очень полезным в связке с возможностью `&allow-other-keys`:

```
(defun foo (size &rest keys &key double &allow-other-keys)
  (let ((v (apply #'make-array size :allow-other-keys t keys)))
    (if double (concatenate (type-of v) v v) v)))

(foo 4 :initial-contents '(a b c d) :double t)
  ⇒ #(a b c d a b c d)
```

[Функция] **funcall** *fn* **&rest** *arguments*

`(funcall` ***fn a1 a2 ... an***`)` применяет функцию *fn* к аргументам *a1*, *a2*, ..., *an*. *fn* не может быть специальной формой или макросом, это не имело бы смысла.

X3J13 voted in June 1988 to allow the *fn* to be only of type `symbol` or `function`; a lambda-expression is no longer acceptable as a functional argument. One must use the `function` special operator or the abbreviation `#'` before a lambda-expression that appears as an explicit argument form.

Например:

```
(cons 1 2) ⇒ (1 . 2)
(setq cons (symbol-function '+))
(funcall cons 1 2) ⇒ 3
```

Различие `funcall` и обычным вызовом функции в том, что функция получается с помощью обычных Lisp вычислений, а не с помощью специальной интерпретации первой позиции в списке формы.

[Константа] **call-arguments-limit**

Значение `call-arguments-limit` положительное целое, которое отображает невключительно максимальное количество аргументов, которые могут быть переданы в функцию. Эта граница зависит от реализации, но не может быть меньше 50. (Разработчикам предлагается сделать этот предел большим на сколько это возможно без ущерба для производительности.) Значение `call-argument-limit` должно, как минимум, равняться `lambda-parameters-limit`. Смотрите также `multiple-values-limit`.

7.4 Последовательное выполнение

Все конструкции в данном разделе выполняют все формы аргументов в прямой последовательности. Различие заключается только в возвращаемых ими результатах.

[Специальный оператор] **progn** {form}*

Конструкция `progn` принимает некоторое количество форм и последовательно вычисляет их слева направо. Значения всех кроме последней формы игнорируется. Результатом формы `progn` становиться то, что вернула последняя форма. Можно сказать, что все формы кроме последней выполняются для побочных эффектов, а последняя форма для значения.

`progn` является примитивной управляющей структурой для «составных выражений», как блоки **begin-end** в Algol'ных языках. Много Lisp'овых конструкций является «неявным `progn`»: так как часть их синтаксиса допускает запись нескольких форм, которые будут выполнены последовательно и возвращён будет результат последней формы.

Если последняя форма `progn` возвращает несколько значение, тогда все они будут возвращены из формы `progn`. Если форм в `progn` нет вообще, то результатом будет `nil`. Эти правила сохраняются также и для неявного `progn`.

[Макрос] **prog1** first {form}*

`prog1` похожа на `progn` за исключением того, что он возвращает результат *первой* формы. Все формы-аргументы выполняются последовательно. Значение первой формы сохраняется, затем выполняются все формы, и, наконец, возвращается сохранённое значение.

`prog1` чаще всего используется, когда необходимо вычислить выражение с побочными эффектами и возвращаемое значение должно быть вычислено *до* побочных эффектов. Например:

```
(prog1 (car x) (rplaca x 'foo))
```

изменяет *car* от `x` на `foo` и возвращает старое значение.

`prog1` всегда возвращает одно значение, даже если первая форма возвращает несколько значений. В следствие, `(prog1 x)` и `(progn x)` могут вести себя по-разному, x возвращает несколько значений. Смотрите `multiple-value-prog1`. И хотя `prog1` может использоваться для явного указания возврата только одного значения, для этих целей лучше использовать функцию `values`.

[Макрос] **prog2** first second {form}*

`prog2` похожа на `prog1`, но она возвращает значение её *второй* формы. Все формы-аргументы выполняются последовательно. Значение второй формы сохраняется и возвращается после выполнения остальных форм. `prog2` представлена по большей части по историческим причинам.

(prog2 a b c ... z) $\equiv$ (progn a (prog1 b c ... z))

Иногда необходимо получить один побочный эффект, затем полезный результат, затем другой побочный эффекта. В таком случае `prog2` полезна. Например:

```
(prog2 (open-a-file) (process-the-file) (close-the-file))
                                ;возвращаемое значение process-the-file
```

7.5 Установка новых связываний переменных

В течение вызова функции представленной лямбда-выражением (или замыканием лямбда-выражения возвращённым функцией `function`), для переменных параметров лямбда-выражения устанавливаются новые связывания. Эти связывания первоначально имеют значения установленные с помощью протокола связывания параметров, описанного в 5.2.2.

Для установки связываний переменных, обычных и функциональных, также полезны следующие конструкции.

[Специальный оператор] **let** ({var | (var [value])}*) {declaration}* {form}*

Форма `let` может быть использована для связи множества переменных со значениями соответствующего множества форм.

Если быть точнее, форма

(let ((*var1 value1*)
(*var2 value2*)
...
(*varm valuem*))
declaration1
declaration2
...
declarationp
body1
body2
...
bodyn)

сначала последовательно выполняет выражения *value1*, *value2* и т.д., сохраняя результаты. Затем все переменные *varj* параллельно привязываются к сохранённым значениям. Каждое связывание будет является лексическим, кроме тех, для которых указана декларация `special`. Затем последовательно выполняются выражения *bodyk*. Все из значения, кроме последнего, игнорируются (другими словами, тело `let` является неявным `progn`). Форма `let` возвращает значение *bodyn* (если тело пустое, что в принципе бесполезно, то `let` возвращает `nil`). Связывания переменных имеют лексическую область видимости и неограниченную продолжительность.

Вместо списка (*varj valuej*), можно записать просто *varj*. В таком случае *varj* инициализируется значением `nil`. В целях хорошего стиля рекомендуется, записывать *varj* только, если в неё будет что-нибудь записано (с помощью `setq` например), перед первым использованием. Если важно, чтобы первоначальное значение было `nil`, вместо некоторого неопределённого значения, тогда будет лучше записать (*varj* `nil`) или (*varj* `'()`), если значение должно обозначать пустой список. Обратите внимание, что код

```
(let (x)
  (declare (integer x))
  (setq x (gcd y z))
  ...)
```

неправильный. Так как x объявлен без первоначального значения и также объявлено, что x это целое число, то произойдёт исключение, так как x при связывании получает `nil` значение, которое не принадлежит целочисленному типу.

Декларации могут использоваться в начале тела `let`. Смотрите `declare`.

Смотрите также `destructuring-bind`.

[Специальный оператор] **let*** ({var | (var [value])}*) {declaration}* {form}*

`let*` похожа на `let`, но связывания переменных осуществляются последовательно, а не параллельно. Это позволяет выражениям для значений переменных ссылаться на ранее связанные переменные.

Если точнее, форма

```
(let* ((var1 value1)
       (var2 value2)
       ...
       (varm valuem))
  declaration1
  declaration2
  ...
  declarationp
  body1
  body2
  ...
  bodyn)
```

сначала вычисляет выражение *value1*, затем с этим значением связывает переменную *var1*, затем вычисляет *value2* и связывает с результатом переменную *var2*, и так далее. Затем последовательно вычисляются выражения *bodyj*. Значения всех выражений, кроме последнего, игнорируются. То есть тело формы `let*` является неявным

`progn`. Форма `let*` возвращает результаты вычисления *bodyn* (если тело пустое, что, в принципе, бесполезно, `let*` возвращает `nil`). Связывания переменных имеют лексическую область видимости и неограниченную продолжительность.

Вместо списка (*varj valuej*), можно записать просто *varj*. В таком случае *varj* будет инициализирована в `nil`. В целях стиля, рекомендуется записывать *varj*, только ей будет что-нибудь присвоено с помощью `setq` перед первым использованием. Если необходимо инициализировать переменную значением `nil`, а не неопределённым, лучше писать (*varj* `nil`) для инициализации «ложью» или (*varj* `'()`) для инициализации пустым списком.

В начале тела `let*` могут использоваться декларации. Смотрите `declare`.

[Специальный оператор] **progv** symbols values {form}*

`progv` является специальной формой, которая позволяет создавать связывания одной и более динамических переменных, чьи имена устанавливаются во время выполнения. Последовательность форм (неявный `progn`) выполняется с динамическими переменными, что имена в списке *symbols* связаны с соответствующими значениями в списке *values*. (Если значений меньше, чем переменных, то соответствующие переменные получают соответствующие значения, а оставшиеся остаются без значений. Смотрите `makunbound`. Если значений больше, чем переменных, они игнорируются.) Результатом `progv` является результат последней формы. Связывания динамических переменных упраздняются при выходе из формы `progv`. Списки переменных и значений это вычисляемые значения. Это то, что отличает `progv` от, например, `let`, в которой имена переменных указываются явно в тексте программы.

`progv` полезна, в частности, для написания интерпретаторов языков встраиваемых в Lisp. Она предоставляет управление механизмом связывания динамических переменных.

[Макрос] **flet** ({(name lambda-list
[[{declaration}* | doc-string]] {form}*)}*)
{declaration}* {form}*
[Макрос] **labels** ({(name lambda-list
[[{declaration}* | doc-string]] {form}*)}*)
{declaration}* {form}*
[Макрос] **macrolet** ({(name varlist
[[{declaration}* | doc-string]] {form}*)}*)
{declaration}* {form}*

`flet` может быть использована для определения локальных именованных функций. Внутри тела формы `flet`, имена функций, совпадающие с именами определёнными в `flet`, ссылаются на локально определённые функции, а не на глобальные определения функции с теми же именами.

Может быть определено любое количество функций. Каждое определение осуществляется формате, как в форме `defun`: сначала имя, затем список параметров (который может содержать `&optional`, `&rest` или `&key` параметры), затем необязательные декларации и строка документации, и, наконец, тело.

```
(flet ((safesqrt (x) (sqrt (abs x))))
  ;; Функция safesqrt используется в двух местах.
  (safesqrt (apply #'+ (map 'list #'safesqrt longlist))))
```

Конструкция `labels` идентична по форме конструкции `flet`. Эти конструкции различаются в том, что область видимости определённых функций для `flet` заключена только в теле, тогда как видимость в `labels` охватывает даже определения этих функций. Это значит, что `labels` может быть использована для определения взаимно рекурсивных функций, а `flet` не может. Это различие бывает полезно. Использование `flet` может локально переопределить глобальную функцию, и новое определение может ссылаться на глобальное. Однако такая же конструкция `labels` не будет обладать этим свойством.

```
(defun integer-power (n k)       ; Быстрое возведение
  (declare (integer n))          ; целого числа в степень
  (declare (type (integer 0 *) k))
```

```
(labels ((expt0 (x k a)
          (declare (integer x a) (type (integer 0 *) k))
          (cond ((zerop k) a)
                ((evenp k) (expt1 (* x x) (floor k 2) a))
                (t (expt0 (* x x) (floor k 2) (* x a)))))
         (expt1 (x k a)
          (declare (integer x a) (type (integer 1 *) k))
          (cond ((evenp k) (expt1 (* x x) (floor k 2) a))
                (t (expt0 (* x x) (floor k 2) (* x a))))))
  (expt0 n k 1)))
```

`macrolet` похожа на форму `flet`, но определяет локальные макросы, используя тот же формат записи, что и `defmacro`. Имена для макросов, установленные с помощью `macrolet`, имеют лексическую область видимости.

I have observed that, while most Common Lisp users pronounce `macrolet` to rhyme with "silhouette," a small but vocal minority pronounce it to rhyme with "Chevrolet." A very few extremists furthermore adjust their pronunciation of `flet` similarly: they say "flay." Hey, hey! *Très outré.*

Макросы часто должны быть раскрыты во «время компиляции» (общими словами, во время перед тем, как сама программа будет выполнена), таким образом, значения переменных во время выполнения не доступны для макросов, определённых с помощью `macrolet`.

X3J13 voted in March 1989 to retract the previous sentence and specify that the macro-expansion functions created by `macrolet` are defined in the lexical environment in which the `macrolet` form appears, not in the null lexical environment. Declarations, `macrolet` definitions, and `symbol-macrolet` definitions affect code within the expansion functions in a `macrolet`, but the consequences are undefined if such code attempts to refer to any local variable or function bindings that are visible in that lexical environment.

Однако, сущности, имеющие лексическую область видимости, *видны* внутри тела формы `macrolet` и *видны* в коде, который является результатом раскрытия макровызова. Следующий пример должен помочь в понимании:

```
;;; Пример macrolet.

(defun foo (x flag)
  (macrolet ((fudge (z)
               ;;Параметры x и flag в данной точке
               ;; недоступны; ссылка на flag была бы
               ;; одноимённую глобальную переменную.
               `(if flag
                    (* ,z ,z)
                    ,z)))
    ;;Параметры x и flag доступны здесь.
    (+ x
       (fudge x)
       (fudge (+ x 1)))))
```

Тело данного примера после разворачивания макросов превращается в

```
(+ x
   (if flag
       (* x x)
       x))
   (if flag
       (* (+ x 1) (+ x 1))
       (+ x 1)))
```

`x` и `flag` легитимно ссылаются на параметры функции `foo`, потому что эти параметры видимы в месте макровызова.

X3J13 voted in March 1988 to specify that the body of each function or expander function defined by `flet`, `labels`, or `macrolet` is implicitly enclosed in a `block` construct whose name is the same as the *name* of the function. Therefore `return-from` may be used to exit from the function.

X3J13 voted in March 1989 to extend `flet` and `labels` to accept any function-name (a symbol or a list whose *car* is `setf`—see section 7.1) as a *name* for a function to be locally defined. In this way one can create local definitions for `setf` expansion functions. (X3J13 explicitly declined to extend `macrolet` in the same manner.)

X3J13 voted in March 1988 to change `flet`, `labels`, and `macrolet` to allow declarations to appear before the body. The new descriptions are therefore as follows:

[Специальный оператор] **symbol-macrolet** ({(var expansion)}*) {declaration}* {form}*

X3J13 проголосовал в июне 1988 адаптировать Common Lisp'овую систему объектов (CLOS). Часть этого является общий механизм, `symbol-macrolet`, для обработки заданных имён переменным, как если бы они были макровызовами без параметров. Эта функциональность полезно независимо от CLOS.

Формы *forms* выполняются как неявный `progn` в лексическом окружении, в котором любая ссылка на обозначенную переменную *var* будет заменена на соответствующее выражение *expansion*. Это происходит, как будто ссылка на переменную *var* является макровызовом без параметров. Выражение *expansion* вычисляется или обрабатывается в месте появления ссылки. Однако, следует отметить, что имена таких макросимволов работает в пространстве имен переменных, не в пространстве функций. Использование `symbol-macrolet` может быть в свою очередь перекрыто с помощью `let` или другой конструкцией, связывающей переменные. Например:

```
(symbol-macrolet ((pollyanna 'goody))
  (list pollyanna (let ((pollyanna 'two-shoes)) pollyanna)))
⇒ (goody two-shoes), not (goody goody)
```

Выражение *expansion* для каждой переменной *var* вычисляется не во время связывания, а во время подстановки вместо ссылок на *var*. Конструкция возвращает значения последней вычисленной формы, или `nil`, если таких значений не было.

Смотрите документация `macroexpand` и `macroexpand-1`. Они раскрывают макросы символов, также как и обычные макросы.

Указанные декларации *declarations* перед телом обрабатываются так как описано в разделе 9.1.

[Макрос] **define-symbol-macro** symbol {form}

Символ *symbol* выступает в качестве макровызова. Одинарная форма

содержит тело раскрытия. Символы не может являться определённой специальной переменной.

7.6 Операторы условных переходов

Традиционная условная конструкция в Lisp'е это `cond`. Однако, `if` гораздо проще и очень похожа на условные конструкции в других языках программирования. Она сделана примитивом в Common Lisp'е. Common Lisp также предоставляет конструкции диспетчеризации (распределения) `case` и `typecase`, которые часто более удобны, чем `cond`.

[Специальный оператор] **if** test then [else]

Специальная формы `if` обозначает то же, что и конструкция **if-then-else** в большинстве других языках программирования. Сначала выполняется форма *test*. Если результат не равен `nil`, тогда выбирается форма *then*. Иначе выбирается форма *else*. Выбранная ранее форма выполняется, и `if` возвращает то, что вернула это форма.

(if *test then else*) ≡ (cond (*test then*) (`t` *else*))

Но в некоторых ситуациях `if` оказывается более читабельным.

Форма *else* может быть опущена. В таком случае, если значение формы *test* является `nil`, тогда ничего не будет выполнено и возвращаемое значение формы `if` будет `nil`. Если в этой ситуации значение формы `if` важно, тогда в зависимости от контекста стилистически удобнее использовать форму `and`. Если значение не важно, тогда удобнее использовать конструкцию `when`.

[Макрос] **when** test {form}*

`(when` *`test form1 form2 ...`* `)` сначала выполняет *test*. Если результат `nil`, тогда ничего не выполняется и возвращается `nil`. Иначе, последовательно выполняются формы *form* слева направо (как неявный `progn`), и возвращается значение последней формы.

(when p a b c) ≡ (and p (progn a b c))
(when p a b c) ≡ (cond (p a b c))

(when p a b c) ≡ (if p (progn a b c) `nil`)
(when p a b c) ≡ (unless (not p) a b c)

В целях хорошего стиля, `when` обычно используется для выполнения побочных эффектов при некоторых условиях, и значение `when` не используется. Если значение все-таки важно, тогда, может быть, стилистически функции `and` или `if` более подходят.

[Макрос] **unless** test {form}*
`(unless` *`test form1 form2 ...`* `)` сначала выполняет *test*. Если результат *не* `nil`, тогда ничего не выполняется и возвращается `nil`. Иначе, последовательно выполняются формы *form* слева направо (как неявный `progn`), и возвращается значение последней формы.

(unless p a b c) ≡ (cond ((not p) a b c))
(unless p a b c) ≡ (if p `nil` (progn a b c))
(unless p a b c) ≡ (when (not p) a b c)

В целях хорошего стиля, `unless` обычно используется для выполнения побочных эффектов при некоторых условиях, и значение `unless` не используется. Если значение все-таки важно, тогда может быть стилистически более подходящая функция `if`.

[Макрос] **cond** {(test {form}*)}*
Форма `cond` содержит некоторое (возможно нулевое) количество подвыражений, которые является списками форм. Каждое подвыражение содержит форму условия и ноль и более форм для выполнения. Например:

(cond (*test-1 consequent-1-1 consequent-1-2 ...*)
(*test-2*)
(*test-3 consequent-3-1 ...*)
...)

Отбирается первое подвыражение, чья форма условия вычисляется в не-`nil`. Все остальные подвыражения игнорируются. Формы отобранного подвыражения последовательно выполняются (как неявный `progn`).

Если быть точнее, `cond` обрабатывает свои подвыражения слева направо. Для каждого подвыражения, вычисляется форма условия. Если результат `nil`, `cond` переходит к следующему подвыражению. Если результат `t`, *cdr* подвыражения обрабатывается, как список форм. Этот список выполняется слева направо, как неявный `progn`. После выполнения списка форм, `cond` возвращает управление без обработки оставшихся подвыражений. Специальная форма `cond` возвращает результат выполнения последней формы из списка. Если этот список пустой, тогда возвращается значение формы условия. Если `cond` вернула управление без вычисления какой-либо ветки (все условные формы вычислялись в `nil`), возвращается значение `nil`.

Для того, чтобы выполнить последнее подвыражение, в случае если раньше ничего не выполнилось, можно использовать `t` для формы условия. В целях стиля, если значение `cond` будет для чего-то использоваться, желательно записывать последнее выражение так: `(t nil)`. Также вопросом вкуса является запись последнего подвыражения `cond` как «синглтон», в таком случае, используется неявный `t`. (Следует отметить, если x может возвращать несколько значений, то `(cond ... (`x`))` может вести себя отлично от `(cond ... (t `x`))`. Первое выражение всегда возвращает одно выражение, тогда как второе возвращает все то же, что и x. В зависимости от стиля, можно указывать поведение явно `(cond ... (t (values `x`)))`, используя функцию `values` для явного указания возврата одного значения.) Например:

```
(setq z (cond (a 'foo) (b 'bar)))     ;Возможна неопределённость
(setq z (cond (a 'foo) (b 'bar) (t nil)))  ;Уже лучше
(cond (a b) (c d) (e))                ;Возможна неопределённость
(cond (a b) (c d) (t e))              ;Уже лучше
(cond (a b) (c d) (t (values e)))     ;Неплохо (если необходимо
                                      ; одно значение)
(cond (a b) (c))                      ;Возможна неопределённость
(cond (a b) (t c))                    ;Уже лучше
(if a b c)                            ;Тоже неплохо
```

Lisp'овая форма `cond` сравнима с последовательностью **if-then-else**, используемой в большинстве алгебраических языках программирования:

(cond (p ...)		**if** p **тогда** ...
(q ...)	roughly	**иначе если** q **тогда** ...
(r ...)	corresponds	**иначе если** r **тогда** ...
...	to	...
(`t` ...))		**иначе** ...

[Макрос] **case** keyform {({({key}*) | key} {form}*)}*

`case` условный оператор, который выбирает ветку для выполнения, в зависимости от равенства некоторой переменной некоторой константе. Константа обычно представляет собой ключевой символ, целое число или строковый символ (но может быть и любой другой объект). Вот развёрнутая форма:

(case *keyform*
 (*keylist-1 consequent-1-1 consequent-1-2* ...)
 (*keylist-2 consequent-2-1* ...)
 (*keylist-3 consequent-3-1* ...)
 ...)

Структурно `case` очень похож на `cond`, и поведение такое же: выбрать список форм и выполнить их. Однако `case` отличается механизмом выбора подвыражения.

Сперва `case` вычисляет форму *keyform* для получения объекта, который называется *ключевой объект*. Затем `case` рассматривает все подвыражения. Если *ключевой объект* присутствует в списке *keylist* (то есть, если *ключевой объект* равен `eql` хотя бы одному элементу из списка *keylist*), то список форм выбранного подвыражения вычисляется, как неявный `progn`. `case` возвращает то же, что и последняя форма списка (или `nil` если список форм был пустой). Если ни одно подвыражение не удовлетворило условию, то `case` возвращает `nil`.

Ключи в списке ключей *keylist не* выполняются. В данном списке должны быть указаны литеральные ключи. Если один ключ попадается более чем в одном подвыражении, это считается ошибкой. Следствием

является то, что порядок этих подвыражений не влияет на поведение конструкции `case`.

Вместо *keylist* можно записать один из символов: `t` или `otherwise`. Подвыражение с одним из таких символов всегда удовлетворяет условию выбора. Такое подвыражение должно быть последним (это исключение из правила о произвольности положения подвыражений). Смотрите также `ecase` и `ccase`, каждая из которых предоставляет неявное `otherwise` подвыражение для сигнализирования об ошибке, если ни одно подвыражение не удовлетворило условию.

Если в подвыражении только один ключ, тогда этот ключ может быть записан вместо списка. Такой «синглтоновый ключ» не может быть `nil` (так как возникают конфликты с (), который означает список без ключей), `t`, `otherwise` или cons-ячейкой.

[Макрос] **typecase** keyform {(type {form}*)}*

`typecase` условный оператор, который выбирает подвыражение на основе типа объекта. Развёрнутая форма:

(typecase *keyform*
 (*type-1 consequent-1-1 consequent-1-2* ...)
 (*type-2 consequent-2-1* ...)
 (*type-3 consequent-3-1* ...)
 ...)

Структура `typecase` похожа на `cond` или `case`. Поведение также схоже в том, что выбирается подвыражение в зависимости от условия. Различие заключается в механизме выбора подвыражения.

Сперва `typecase` вычисляет форму *keyform* для создания объекта, называемого ключевым объектом. Далее `typecase` друг за другом рассматривает каждое подвыражение. Форма *type*, которая встречается в каждом подвыражении, является спецификатором типа. Данный спецификатор не вычисляется, поэтому должен быть литеральным. Когда ключевой объект принадлежит некоторый типу, то выделенный список форм *consequent* выполняется последовательно (как неявный `progn`). `typecase` возвращает то, что вернула последняя форма из списка (или `nil` если список был пуст). Если не одно подвыражение не было выбрано, `typecase` возвращает `nil`.

Как и для `case` можно использовать `t` или `otherwise` на позиции *типа* для задания подвыражений, которые будут выполняться, только если не было выполнено других подвыражений. Смотрите также `etypecase` и `ctypecase`, каждая из которых предоставляет неявную ветку `otherwise` для сигнализирования об ошибке, что ни одно подвыражение не удовлетворило условию.

Допустимо указывать более одного подвыражение, тип условия которого уже является подтипом условия другого подвыражения. В таком случае будет выбрано первое встретившееся подвыражение. Таким образом в `typecase`, в отличие от `case`, порядок следования подвыражений влияет на поведение всей конструкции.

```
(typecase an-object
  (string ...)          ;Подвыражение обрабатывает строки
  ((array t) ...)       ;Подвыражение обрабатывает общие массивы
  ((array bit) ...)     ;Подвыражение обрабатывает битовые массивы
  (array ...)           ;Обрабатывает все остальные массивы
  ((or list number) ...);Подвыражение обрабатывает списки и числа
  (t ...))              ;Подвыражение обрабатывает все остальные объекты
```

7.7 Блоки и выходы

Конструкции `block` и `return-from` предоставляют функциональность для структурированного лексического нелокального выхода. В любом месте лексически внутри конструкции `block`, для мгновенного возврата управления из `block` может быть использована `return-from` с тем же именем. В большинстве случаев этот механизм более эффективный, чем функциональность динамического нелокального выхода, предоставляемая формами `catch` и `throw`, описанными в разделе 7.11.

[Специальный оператор] **block** name {form}*

Конструкция `block` слева направо выполняет каждую форму *form*, возвращая то, что возвращает последняя форма. Если, однако, в процессе выполнения форм, будет выполнена `return` или `return-from`

с именем *name*, тогда будет возвращён результат заданный одной из этих форм, и поток выполнения немедленно выйдет из формы `block`. Таким образом `block` отличается от `progn` тем, что последняя никак не реагирует на `return`.

Имя блока не выполняется. Оно должно быть символом. Область видимости имени блока лексическая. Из блока можно осуществить выход с помощью `return` или `return-from`, только если они содержаться в тексте в блоке. Продолжительность видимости имени динамическая. Таким образом из блока во время выполнения можно выйти только один раз, обычно или явно с помощью `return`.

Форма `defun` неявно помещает тело функции в одноимённый блок. Таким образом можно использовать `return-from` для преждевременного выхода из функции в определении `defun`.

Лексическая область видимости имени блока полноценна и имеет последствия, которые могут быть сюрпризом для пользователей и разработчиков других Lisp систем. Например, `return-from` в следующем примере в Common Lisp работает так как и ожидается:

```
(block loser
  (catch 'stuff
    (mapcar #'(lambda (x) (if (numberp x)
                              (hairyfun x)
                              (return-from loser nil)))
            items)))
```

В зависимости от ситуации, `return` в Common Lisp'е может быть не проста. `return` может перескочить ловушки, если это необходимо, для рассматриваемого блока. Также возможно для «замыкания», созданного с помощью `function` для лямбда-выражения, ссылаться на имя блока на протяжении лексической доступности этого блока.

[Специальный оператор] **return-from** name [result]

`return-from` используется для возврата из `block` или из таких конструкций, как `do` и `prog`, которые неявно устанавливают `block`. Имя *name* не выполняется и должно быть символом. Конструкция `block` с этим именем должна лексически охватывать форму `return-from`. Каков бы ни был результат вычисления формы *result*, управление немедленно

возвращается из блока. (Если форма *result* опущена, тогда используется значение `nil`. В целях стиля, эта форма обязана использоваться для указания того, что возвращаемое значение не имеет ценности.)

Форма `return-from` сама по себе ничего и никогда не возвращает. Она указывает на то, что результат выполнения будет возвращён из конструкции `block`. Если вычисление формы *result* приводит к нескольким значение, эти несколько значений и будут возвращены и конструкции.

[Макрос] **return** [result]

`(return` *form*`)` идентично по смыслу `(return-from nil` *form*`)`. Она возвращает управление из блока с именем `nil`. Такие блоки с именем /nil устанавливаются автоматически в конструкциях циклов, таких как `do`, таким образом `return` будет производить корректный выход из таких конструкций.

7.8 Формы циклов

Common Lisp предоставляет некоторые конструкции для циклов. Конструкция `loop` предоставляет простую функциональность. Она слегка больше, чем `progn`, и имеет ветку для переноса управления снизу вверх. Конструкции `do` и `do*` предоставляют общую функциональность для управления на каждом цикле изменением нескольких переменных. Для специализированных циклов над элементами списка или n последовательных чисел предоставляются формы `dolist` и `dotimes`. Конструкция `tagbody` наиболее общая конструкция, которая внутри себя позволяет использование выражений `go`. (Традиционная конструкция `prog` — это синтез `tagbody`, `block` и `let`.) Большинство конструкций циклов позволяют определённые статически нелокальные выходы (смотрите `return-from` и `return`).

7.8.1 Бесконечный цикл

Конструкция `loop` является наипростейшей функциональностью для итераций. Она не управляет переменными, и просто циклично

выполняет своё тело.

[Макрос] **loop** {form}*

Каждая форма *form* выполняется последовательно слева направо. Когда вычислена последняя форма, тогда вычисляется первая форма и так далее, в безостановочном цикле. Конструкция `loop` никогда не возвращает значение. Её выполнение может быть остановлено явно, с помощью `return` или `throw`, например.

`loop`, как и многие конструкции циклов, устанавливает неявный блок с именем `nil`. Таким образом, `return` с заданным результатом может использоваться для выхода и `loop`.

7.8.2 Основные формы циклов

В отличие от `loop`, `do` и `do*` предоставляют мощный механизм для повторных вычислений большого количества переменных.

[Макрос] **do** ({var | (var [init [step]])}*)
(end-test {result}*)
{declaration}* {tag | statement}*
[Макрос] **do*** ({var | (var [init [step]])}*)
(end-test {result}*)
{declaration}* {tag | statement}*

Специальная форма `do` представляет общую функциональность цикла, с произвольным количеством «переменных-индексов». Эти переменные связываются при входе в цикл и параллельно наращиваются, как это было задано. Они могут быть использованы, как для генерации необходимых последовательных чисел (как, например, последовательные целые числа), так и для накопления результата. Когда условие окончания цикла успешно выполнилось, тогда цикл завершается с заданным значением.

В общем виде `do` выглядит так:

(do ((*var1 init1 step1*)
(*var2 init2 step2*)
...
(*varn initn stepn*))

(*end-test* . *result*)
{declaration}*
. *tagbody*)

Цикл `do*` выглядит также, кроме изменения имени с `do` на `do*`.

Первый элемент формы является списком нуля и более спецификаторов переменных-индексов. Каждый спецификатор является списком из имени переменной *var*, первоначального значения *init*, и форма приращения *step*. Если *init* опущен, используется первоначальное значение `nil`. Если *step* опущен, *var* не изменяется на итерациях цикла (но может изменяться в теле цикла с помощью формы `setq`).

Спецификатор переменной-индекса может также быть просто именем переменной. В этом случае переменная будет иметь первоначальное значение `nil` и не будет изменяться при итерациях цикла. В целях стиля, использовать просто имя переменной рекомендуется, только если перед первым использованием для неё устанавливается значение с помощью `setq`. Если необходимо чтобы первоначальное значение было `nil`, а не неопределённое, то лучше указывать это явно, если нужна ложь так: `(`*`varj`* `nil)` или если нужен пустой список так: `(`*`varj`* `'())`.

Перед первой итерацией вычисляются все формы *init*, и каждая переменная *var* связывается с соответствующим результатом вычислений *init*. Используется именно связывание, а не присвоение. Когда цикл завершается, старые значения этих переменных восстанавливаются. Для `do`, *все* формы *init* вычисляется перед тем, как будут связаны переменные *var*. Таким образом все формы могут ссылаться на старые связывания этих переменных (то есть на значения, которые были видимы до начала выполнения конструкции `do`). Для `do*` вычисляется первая форма *init*, затем первая переменная связывается с результатом этих вычислений. Затем вычисляется вторая форма *init* и вторая переменная *var* связывается с этим значением, и так далее. В целом, форма *initj* может ссылаться на *новые* связывания *vark*, если $k < j$, иначе ссылка происходит на *старое* связывание.

Второй элемент конструкции цикла это список из формы предиката-выхода *end-test* и нуля и более форм результата *result*. Этот элемент напоминает подвыражение `cond`. В начале каждой итерации, после обработки всех переменных, вычисляется форма *end-test*. Если результат `nil`, выполняется тело формы `do` (или `do*`). Если результат не

`nil`, последовательно вычисляются формы *result*, как неявный `progn`, и затем `do` возвращает управление. `do` возвращает результаты вычисления последней формы *result*. Если таких форм не быть, значением `do` становиться `nil`. Следует отметить, что аналогия с подвыражениями `cond` не полная, так как `cond` в этом случае возвращает результат формы условия.

Переменные-индексы изменяются в начале каждой непервой итерации так, как написано далее. Слева направо вычисляются все формы *step*, и затем результаты присваиваются переменным-индексам. Если такой формы *step* для переменной указано не было, то переменная и не изменяется. Для `do`, все формы *step* вычисляются перед там, как будут изменены переменные. Присваивания переменным осуществляются параллельно, как в `psetq`. Так как *все* формы *step* вычисляются перед тем, как будет изменена хоть одна переменных, форма *step* при вычислении всегда ссылается на старые значения *всех* переменных-индексов, даже если другие формы *step* были выполнены. Для `do*`, вычисляется первая форма *step*, затем полученное значение присваивается первой переменной-индексом, затем вычисляется вторая форма *step*, и полученное значение присваивается второй переменной, и так далее. Присваивание происходит последовательно, как в `setq`. И для `do`, и для `do*` после того как переменные были изменены, вычисляется *end-test* так, как уже было описано выше. Затем продолжаются итерации.

Если *end-test* формы `do` равен `nil`, тогда предикат всегда ложен. Таким образом получается «бесконечный цикл»: тело *body* `do` выполняется циклично, переменные-индексы изменяются как обычно. (Конструкция `loop` также является «бесконечным циклом», только без переменных-индексов.) Бесконечный цикл может быть остановлен использованием `return`, `return-from`, `go` на более высокий уровень или `throw`. Например:

```
(do ((j 0 (+ j 1)))
    (nil)                       ;Выполнять вечно
  (format t "~%Input ~D:" j)
  (let ((item (read)))
    (if (null item) (return)    ;Обрабатывать элементы пока не найден nil
        (format t "~&Output ~D: ~S" j (process item)))))
```

Оставшаяся часть `do` оборачивается в неявный `tagbody`. Теги могут использоваться внутри тела цикла `do` для того, чтобы затем использовать выражения `go`. На такие выражения `go` не могут использоваться в спецификаторах переменных-индексов, в предикате *end-test* и в формах результата *result*. Когда управление достигает конца тела цикла `do`, наступает следующая цикл итерации (начинающийся с вычисления форм *step*).

Неявный `block` с именем `nil` окружает всю форму `do`. Выражение `return` может использоваться в любом месте для немедленного выхода из цикла.

Формы `declare` могут использоваться в начала тела `do`. Они применяются к коду внутри тела `do`, для связываний переменных-индексов, для форм *init*, для форм *step*, для предиката *end-test* и для форм результата *result*.

Вот парочка примеров использования `do`:

```
(do ((i 0 (+ i 1))     ;Sets every null element of a-vector to zero
     (n (length a-vector)))
    ((= i n))
  (when (null (aref a-vector i))
    (setf (aref a-vector i) 0)))
```

Конструкция

```
(do ((x e (cdr x))
     (oldx x x))
    ((null x))
  body)
```

использует параллельное присваивание переменным-индексам. На первой итерации значение `oldx` получает значение `x`, которое было до входа в цикл. При выходе из цикла `oldx` будет содержать значение `x`, которое было на предыдущей итерации.

Очень часто алгоритм цикла может быть по большей части выражен в формах *step* и тело при этом останется пустым. Например,

```
(do ((x foo (cdr x))
     (y bar (cdr y))
     (z '() (cons (f (car x) (car y)) z)))
    ((or (null x) (null y))
     (nreverse z)))
```

делает то же, что и `(mapcar #'f foo bar)`. Следует отметить, что вычисление *step* для `z` использует тот факт, что переменные переприсваиваются параллельно. Тело функции пустое. Наконец, использование `nreverse` в форме возврата результата, переставляет элементы списка для правильного результата. Другой пример:

```
(defun list-reverse (list)
       (do ((x list (cdr x))
            (y '() (cons (car x) y)))
           ((endp x) y)))
```

Нужно заметить, что используется `endp` вместо `null` или `atom` для проверки конца списка. Это даёт более надёжный алгоритм.

В качестве примера вложенных циклов, предположим что `env` содержит список cons-ячеек. *car* элемента каждой cons-ячейки является списком символов, и *cdr* каждой cons-ячейки является списком такой же длины с соответствующими значениями. Такая структура данных похожа на ассоциативный список, но она в отличие разделена на «кадры». Общая структура напоминает грудную клетку. Функция поиска по такой структуре может быть такой:

```
(defun ribcage-lookup (sym ribcage)
       (do ((r ribcage (cdr r)))
           ((null r) nil)
         (do ((s (caar r) (cdr s))
              (v (cdar r) (cdr v)))
             ((null s))
           (when (eq (car s) sym)
             (return-from ribcage-lookup (car v))))))
```

(Примечание, использование отступов в примере выше выделяет тела вложенных циклов.)

Цикл `do` может быть выражен в терминах более примитивных конструкций `block`, `return`, `let`, `loop`, `tagbody` и `psetq`:

```
(block nil
  (let ((var1 init1)
        (var2 init2)
        ...
        (varn initn))
    {declaration}*
    (loop (when end-test (return (progn . result)))
          (tagbody . tagbody)
          (psetq var1 step1
                 var2 step2
                 ...
                 varn stepn))))
```

`do*` почти то же, что и `do` за исключением того, что связывание и наращение переменных происходит последовательно, а не параллельно. Таким образом, в вышеприведённой конструкции, `let` будет заменена на `let*` и `psetq` на `setq`.

7.8.3 Простые формы циклов

Конструкции `dolist` и `dotimes` для каждого значения взятого для одной переменной выполняют тело один раз. Они хоть и выражаются в терминах `do`, но захватывают очень простые шаблоны использования.

И `dolist` и `dotimes` циклично выполняют тело. На каждой итерации заданная переменная связывается с элементом, которая затем может использоваться в теле. `dolist` использует элементы списка, и `dotimes` использует целые числа от 0 по $n - 1$, при некотором указанном положительном целом n.

Результат двух этих конструкций может быть указан с помощью необязательной формы результата. Если эта форма опущена результат равен `nil`.

Выражение `return` может быть использовано для немедленного возврата из форм `dolist` или `dotimes`, игнорируя все оставшиеся итерации, которые должны были быть выполнены. `block` с именем `nil` окружает конструкцию. Тело цикла неявно обернуто конструкцией `tagbody`. Таким образом тело может содержать теги и `go` выражения. Декларации могут быть указаны перед телом цикла.

[Макрос] **dolist** (var listform [resultform])
{declaration}* {tag | statement}*

`dolist` предоставляет прямой цикл по списку элементов. Сначала `dolist` вычисляет форму *listform*, которая должна вернуть список. Затем для каждого элемента вычисляется тело цикла. Данный элемент на каждой итерации связывается с переменной *var*. Затем вычисляется *resultform* (одна форма, *не* неявный `progn`). (Когда вычисляется *resultform*, переменная *var* все ещё связана, и имеет значение `nil`.) Если *resultform* опущена, то результат равен `nil`.

(dolist (x '(a b c d)) (prin1 x) (princ " ")) ⇒ `nil`
вывод «`a b c d` » (в том числе пробел в конце)

Для завершения цикла и возврата заданного значения может использоваться явное выражение `return`.

[Макрос] **dotimes** (var countform [resultform])
{declaration}* {tag | statement}*

`dotimes` предоставляет цикл над последовательностью целых чисел. Выражение `(dotimes (`*`var countform resultform`*`) .` *`progbody`*`)` вычисляет форму *countform*, которая должна вернуть целое число. Затем тело цикла выполняется по порядку один раз для каждого число от нуля (включительно) до *count* (исключая). При этом переменная *var* связывается с текущим целым числом. Если значение *countform* отрицательно или равно нулю, тогда *progbody* не выполняется ни разу. Наконец выполняется *resultform* (одна форма, *не* неявный `progn`), и полученный результат возвращается из формы цикла. (Когда *result* вычисляется, переменная-индекс *var* все ещё связана и содержит количество выполненных итераций.) Если *resultform* опущена, то результат равен `nil`.

Для завершения цикла и возврата заданного значения может использоваться выражение `return`.

Пример использования `dotimes` для обработки строк:

```
;;; True if the specified subsequence of the string is a
;;; palindrome (reads the same forwards and backwards).

(defun palindromep (string &optional
                           (start 0)
                           (end (length string)))
  (dotimes (k (floor (- end start) 2) t)
    (unless (char-equal (char string (+ start k))
                        (char string (- end k 1)))
      (return nil))))

(palindromep "Able was I ere I saw Elba") ⇒ t

(palindromep "A man, a plan, a canal–Panama!") ⇒ nil

(remove-if-not #'alpha-char-p     ;Удалить знаки препинания
               "A man, a plan, a canal–Panama!")
  ⇒ "AmanaplanacanalPanama"

(palindromep
 (remove-if-not #'alpha-char-p
                "A man, a plan, a canal–Panama!")) ⇒ t

(palindromep
 (remove-if-not
   #'alpha-char-p
   "Unremarkable was I ere I saw Elba Kramer, nu?")) ⇒ t

(palindromep
 (remove-if-not
   #'alpha-char-p
   "A man, a plan, a cat, a ham, a yak,
                  a yam, a hat, a canal–Panama!")) ⇒ t
```

```
(palindromep
 (remove-if-not
   #'alpha-char-p
   "Ja-da, ja-da, ja-da ja-da jing jing jing")) ⇒ nil
```

Изменение значения переменной *var* в теле цикла (с помощью `setq` например) будет иметь непредсказуемые последствия, возможно зависящие от реализации. Компилятор Common Lisp'а может вывести предупреждение о том, что переменная-индекс используется в `setq`.

Смотрите также `do-symbols`, `do-external-symbols` и `do-all-symbols`.

7.8.4 Отображение

Отображение — это тип цикла, в котором заданная функция применяется к частям одной или более последовательностей. Результатом цикла является последовательность, полученная из результатов выполнения это функции. Существует несколько опции для указания того, какие части списка будут использоваться в цикле, и что будет происходить с результатом применения функции.

Функция `map` может быть использована для отображения любого типа последовательности. Следующие же функции оперируют только списками.

[Функция] **mapcar** *function list* **&rest** *more-lists*
[Функция] **maplist** *function list* **&rest** *more-lists*
[Функция] **mapc** *function list* **&rest** *more-lists*
[Функция] **mapl** *function list* **&rest** *more-lists*
[Функция] **mapcan** *function list* **&rest** *more-lists*
[Функция] **mapcon** *function list* **&rest** *more-lists*

Для каждой из этих функций отображения, первый аргумент является функцией и оставшиеся аргументы должны быть списками. Функция в первом аргументе должно принимать столько аргументов, сколько было передано списков в функцию отображения.

`mapcar` последовательно обрабатывает элементы списков. Сначала функция применяется к *car* элементу каждого списка, затем к *cadr*

элементу, и так далее. (Лучше всего, чтобы все переданные списки имели одинаковую длину. Если это не так, то цикл завершиться, как только закончится самый короткий список, и все оставшиеся элементы в других списках будут проигнорированы.) Значение, возвращаемое `mapcar`, является списком результатов последовательных вызовов функции из первого параметра. Например:

(mapcar #'abs '(3 -4 2 -5 -6)) ⇒ (3 4 2 5 6)
(mapcar #'cons '(a b c) '(1 2 3)) ⇒ ((a . 1) (b . 2) (c . 3))

`maplist` похожа на `mapcar` за исключением того, что функция применяется к спискам и последующим *cdr* элементам этих списков, а не последовательно к элементам спискам. Например:

```
(maplist #'(lambda (x) (cons 'foo x))
         '(a b c d))
  ⇒ ((foo a b c d) (foo b c d) (foo c d) (foo d))
```

```
(maplist #'(lambda (x) (if (member (car x) (cdr x)) 0 1)))
         '(a b a c d b c))
  ⇒ (0 0 1 0 1 1 1)
  ;Возвращается 1, если соответствующий элемент входящего списка
  ;  появлялся последний раз в данном списке.
```

`mapl` и `mapc` похожи на `maplist` и `mapcar`, соответственно, за исключением того, что они не накапливают результаты вызова функций.

Эти функции используются, когда функция в первом параметре предназначена для побочных эффектов, а не для возвращаемого значения. Значение возвращаемое `mapl` или `mapc` является вторым аргументом, то есть первой последовательностью для отображения.

`mapcan` и `mapcon` похожи на `mapcar` и `maplist` соответственно, за исключением того, что результат создаётся с помощью функции `nconc`, а не `list`. То есть,

(mapcon *f x1 ... xn*)
  ≡ (apply #'nconc (maplist *f x1 ... xn*))

Такое же различие между `mapcan` и `mapcar`. Концептуально, эти функции позволяют функции отображения возвращать переменное количество элементов. Таким образом длина результата может быть не равна длине входного списка. Это, в частности, полезно для возврата нуля или одного элемента:

```
(mapcan #'(lambda (x) (and (numberp x) (list x)))
        '(a 1 b c 3 4 d 5))
  ⇒ (1 3 4 5)
```

В этом случае функция действует, как фильтр. Это стандартная Lisp'овая идиома использования `mapcan`. (Однако, в этом контексте функция `remove-if-not` также может быть полезна.) Помните, что `nconc` деструктивная операция, следовательно и `mapcan` и `mapcon` также деструктивны. Список возвращаемый функцией *function* изменяется для соединения и возврата результата.

Иногда `do` или прямая последовательная рекурсия удобнее, чем функции отображения. Однако, функции отображения должны быть использованы везде, где они действительно необходимы, так как они увеличивают ясность кода.

Функциональный аргумент функции отображения должен быть подходящим для функции `apply`. Он не может быть макросов или именем специальной формы. Кроме того, в качестве функционального аргумента можно использовать функцию, имеющую `&optional` и `&rest` параметры.

7.8.5 Использование «GOTO»

Реализации Lisp'а начиная с Lisp'а 1.5 содержат то, что изначально называлось «the program feature», как будто без этого невозможно писать программы! Конструкция `prog` позволяет писать в Algol- или Fortran- императивном стиле, используя выражения `go`, которые могут ссылаться на теги в теле `prog`. Современный стиль программирования на Lisp'е стремится снизить использование `prog`. Различные конструкции циклов, как `do`, имеют тела с характеристиками `prog`. (Тем не менее, возможность использовать выражения `go` внутри конструкции цикла очень редко используется на практике.)

`prog` предоставляет три различные операции: связывание локальный переменных, использование выражения `return` и использование выражения `go`. В Common Lisp'е эти три операции были разделены на три конструкции: `let`, `block` и `tagbody`. Эти три конструкции могут использоваться независимо, как строительные кирпичики для других типов конструкций.

[Специальный оператор] **tagbody** {tag | statement}*

Часть `tagbody` после списка переменные называется *телом*. Элемент тела может быть символов или целым числом, и называться в этом случае *тег*. Также элемент тела может быть списком, и называться в этом случае *выражением*.

Каждый элемент тела обрабатывается слева направо. *Теги* игнорируются. *Выражения* вычисляются и их результаты игнорируются. Если управление достигает конца тела, `tagbody` возвращает `nil`.

Если вычисляется форма (`go` *`tag`*), управление перемещается на часть тела, обозначенную *тегом*.

Область видимости тегов, устанавливаемых в `tagbody`, является лексической. Продолжительность видимости тегов динамическая. Когда управление вышло из конструкции `tagbody`, ссылка `go` на теги в её теле невозможны. Существует возможность для `go` прыжка в `tagbody`, которая не находится внутри конструкции `tagbody`, которая и содержала этот `go`. То есть возможны прыжки в родительский `tagbody`. Теги устанавливаемые `tagbody` будут только скрывать другие одноимённые теги.

Лексическая область видимости для тегов (целей `go`) полностью полноправно и последствия могут быть сюрпризом для пользователей и разработчиков других Lisp систем. Например, `go` в следующем примере, работает в Common Lisp так, как это и ожидается:

```
(tagbody
  (catch 'stuff
    (mapcar #'(lambda (x) (if (numberp x)
                              (hairyfun x)
                              (go lose)))
            items))
```

```
  (return)
 lose
  (error "I lost big!"))
```

В зависимости от ситуации, `go` в Common Lisp'е не обязательно похож на простую машинную инструкцию «jump». Если необходимо, `go` может перепрыгивать ловушки исключений. Возможно так, что «замыкание», созданное с помощью `function`, для лямбда-выражения ссылается на тег (цель `go`) так долго, сколько лексически доступен данный тег. Смотрите 3 для понимания этого примера.

[Макрос] **prog** ({var | (var [init])}*) {declaration}* {tag | statement}*
[Макрос] **prog*** ({var | (var [init])}*) {declaration}* {tag | statement}*

Конструкция `prog` является синтезом `let`, `block` и `tagbody`, позволяющая связывать переменные, использовать `return` и `go` в одной конструкции. Обычно конструкция `prog` выглядит так:

```
(prog (var1 var2 (var3 init3) var4 (var5 init5))
     {declaration}*
     statement1
 tag1
     statement2
     statement3
     statement4
 tag2
     statement5
     ...
     )
```

Список после ключевого символа `prog` является множеством спецификаторов для связывания переменных *var1*, *var2*. Этот список обрабатывается так же, как и в выражении `let`: сначала слева направо выполняются все формы *init* (если формы нет, берётся значение `nil`), и затем переменные параллельно связываются с полученными ранее значениями. Возможно использовать *декларации* в начале дела `prog` так же, как и в `let`.

Тело `prog` выполняется, как обернутое в `tagbody`. Таким образом, для перемещения управления к *тегу* могут использоваться выражения `go`.

`prog` неявно устанавливает вокруг тела `block` с именем `nil`. Это значит, можно в любое время использовать `return` для выхода из конструкции `prog`.

Вот небольшой пример того, что можно сделать с помощью `prog`:

```
(defun king-of-confusion (w)
  "Take a cons of two lists and make a list of conses.
   Think of this function as being like a zipper."
  (prog (x y z)     ;Инициализировать x, y, z в nil
        (setq y (car w) z (cdr w))
   loop
        (cond ((null y) (return x))
              ((null z) (go err)))
   rejoin
        (setq x (cons (cons (car y) (car z)) x))
        (setq y (cdr y) z (cdr z))
        (go loop)
   err
        (cerror "Will self-pair extraneous items"
                "Mismatch - gleep!  S" y)
        (setq z y)
        (go rejoin)))
```

которые делает то же, что и:

```
(defun prince-of-clarity (w)
  "Take a cons of two lists and make a list of conses.
   Think of this function as being like a zipper."
  (do ((y (car w) (cdr y))
       (z (cdr w) (cdr z))
       (x '() (cons (cons (car y) (car z)) x)))
      ((null y) x)
    (when (null z)
      (cerror "Will self-pair extraneous items"
              "Mismatch - gleep!  S" y)
```

```
    (setq z y))))
```

Конструкция `prog` может быть выражена в терминах более простых конструкций `block`, `let` и `tagbody`:

(prog *variable-list* {declaration}* . *body*)
≡ (block nil (let *variable-list* {declaration}* (tagbody . *body*)))

Специальная форма `prog*` очень похожа на `prog`. Одно отличие в том, что связывание и инициализация переменных осуществляется *последовательно*, тем самым форма *init* использовать значения ранее связанных переменных. Таким образом `prog*` относится к `prog`, как `let*` к `let`. Например,

```
(prog* ((y z) (x (car y)))
      (return x))
```

возвращает *car* элемент значения `z`.

[Специальный оператор] **go** tag

Специальная форма `(go` ***tag*** `)` используется для применения «goto» внутри конструкции `tagbody`. *tag* должен быть символов или целым числом. *tag* не вычисляется. `go` переносит управление на точку тела, которая была помечена тегом равным `eql` заданному. Если такого тега в теле нет, поиск осуществляется в лексически доступном теле другой конструкции `tagbody`. Использоваться `go` с тегом, которого нет, является ошибкой.

Форма `go` никогда не возвращает значение.

В целях стиля, рекомендуется дважды подумать, прежде чем использовать `go`. Большинство функций `go` могут быть заменены циклами, вложенными условными формами или `return-from`. Если использование `go` неизбежно, рекомендуется управляющую структуру реализованную с помощью `go` «упаковать» в определении макроса.

7.9 Structure Traversal and Side Effects

X3J13 voted in January 1989 to restrict side effects during the course of a built-in operation that can execute user-supplied code while traversing a data structure.

Consider the following example:

```
(let ((x '(apples peaches pumpkin pie)))
  (dolist (z x)
    (when (eq z 'peaches)
      (setf (cddr x) '(mango kumquat)))
    (format t " S " (car z))))
```

Depending on the details of the implementation of `dolist`, this bit of code could easily print

```
apples peaches mango kumquat
```

(which is perhaps what was intended), but it might as easily print

```
apples peaches pumpkin pie
```

Here is a plausible implementation of `dolist` that produces the first result:

```
(defmacro dolist ((var listform &optional (resultform 'nil))
                  &body body)
  (let ((tailvar (gensym "DOLIST")))
    `(do ((,tailvar ,listform (cdr ,tailvar)))
         ((null ,tailvar) ,resultform)
       (let ((,var (car ,tailvar))) ,@body))
```

But here is a plausible implementation of `dolist` that produces the second result:

```
(defmacro dolist ((var listform &optional (resultform "nil))
                  &body body)
  (let ((tailvar (gensym "DOLIST")))
    `(do ((,tailvar ,listform))
         ((null ,tailvar) ,resultform)
       (let ((,var (pop ,tailvar))) ,@body))
```

The X3J13 recognizes and legitimizes varying implementation practices: in general it is an error for code executed during a "structure-traversing" operation to destructively modify the structure in a way that might affect the ongoing traversal operation. The committee identified in particular the following special cases.

For list traversal operations, the *cdr* chain may not be destructively modified.

For array traversal operations, the array may not be adjusted (see `adjust-array`) and its fill pointer, if any, may not be modified.

For hash table operations (such as `with-hash-table-iterator` and `maphash`), new entries may not be added or deleted, *except* that the very entry being processed by user code may be changed or deleted.

For package symbol operations (for example, `with-package-iterator` and `do-symbols`), new symbols may not be interned in, nor symbols uninterned from, the packages being traversed or any packages they use, *except* that the very symbol being processed by user code may be uninterned.

X3J13 noted that this vote is intended to clarify restrictions on the use of structure traversal operations that are not themselves inherently destructive; for example, it applies to `map` and `dolist`. Destructive operators such as `delete` require even more complicated restrictions and are addressed by a separate proposal.

The X3J13 vote did not specify a complete list of the operations to which these restrictions apply. Table 7.1 shows what I believe to be a complete list of operations that traverse structures and take user code as a body (in the case of macros) or as a functional argument (in the case of functions).

In addition, note that user code should not modify list structure that might be undergoing interpretation by the evaluator, whether explicitly in-

voked via `eval` or implicitly invoked, for example as in the case of a hook function (a `defstruct` print function, the value of `*evalhook*` or `*applyhook*`, etc.) that happens to be a closure of interpreted code. Similarly, `defstruct` print functions and other hooks should not perform side effects on data structures being printed or being processed by `format`, or on a string given to `make-string-input-stream`. You get the idea; be sensible.

Note that an operation such as `mapcar` or `dolist` traverses not only *cdr* pointers (in order to chase down the list) but also *car* pointers (in order to obtain the elements themselves). The restriction against modification appears to apply to all these pointers.

7.10 Возврат и обработка нескольких значений

Обычно результатом вызова Lisp'овой функции является один Lisp'овый объект. Однако, иногда для функции удобно вычислить несколько объектов и вернуть их. Common Lisp представляет механизм для прямой обработки нескольких значений. Механизм удобнее и эффективнее, чем исполнение трюков со списками или глобальными переменными.

7.10.1 Конструкции для обработки нескольких значений

Обычно не используется несколько значений. Для возврата и обработки нескольких значений требуются специальные формы. Если вызывающий функцию код не требует нескольких значений, однако вызываемая функция их несколько, то для кода берётся только первое значение. Все оставшиеся значения игнорируются. Если вызываемая функция возвращает ноль значений, вызывающий код в качестве значения получает `nil`.

`values` — это главный примитив для возврата нескольких значений. Он принимает любое количество аргументов и возвращает столько же значений. Если последняя форма тела функции является `values` с тремя аргументами, то вызов такой функции вернёт три значения. Другие

специальные формы также возвращают несколько значения, но они могут быть описаны в терминах `values`. Некоторые встроенные Common Lisp функции, такая как `floor`, возвращают несколько значений.

Специальные формы обрабатывающие несколько значений представлены ниже:

multiple-value-list
multiple-value-call
multiple-value-prog1
multiple-value-bind
multiple-value-setq

Они задают форму для вычисления и указывают куда поместить значения возвращаемые данной формой.

[Функция] **values** **&rest** *args*

Все аргументы в таком же порядке возвращаются, как значения. Например,

```
(defun polar (x y)
  (values (sqrt (+ (* x x) (* y y))) (atan y x)))

(multiple-value-bind (r theta) (polar 3.0 4.0)
  (vector r theta))
  ⇒ #(5.0 0.9272952)
```

Выражение `(values)` возвращает ноль значений. Это стандартная идиома для возврата из функции нулевого количества значений.

Иногда необходимо указать явно, что функция будет возвращать только одно значение. Например, функция

```
(defun foo (x y)
  (floor (+ x y) y))
```

будет возвращать два значения, потому что `floor` возвращает два значения. Может быть второе значение не имеет смысла в данном контексте, или есть причины не тратить время на вычисления второго значения. Функция `values` является стандартной идиомой для указания того, что будет возвращено только одно значение, как показано в следующем примере.

```
(defun foo (x y)
  (values (floor (+ x y) y)))
```

Это работает, потому что `values` возвращает только *одно* значения для каждой формы аргумента. Если форма аргумента возвращает несколько значений, то используется только первое, а остальные игнорируются.

В Common Lisp'е для вызывающего кода нет возможности различить, было ли возвращено просто одно значение или было возвращено только одно значение в с помощью `values`. Например значения, возвращённые выражением `(+ 1 2)` и `(values (+ 1 2))`, идентичны во всех отношениях: они оба просто равны `3`.

[Константа] **multiple-values-limit**

Значение `multiple-values-limit` является положительным целым числом, которое невключительно указывает наибольшее возможное количество возвращаемых значений. Это значение зависит от реализации, но не может быть менее 20. (Разработчики рекомендуется делать это ограничение как можно большим без потери в производительности.) Смотрите `lambda-parameters-limit` и `call-arguments-limit`.

[Функция] **values-list** *list*

Все элементы списка *list* будут возвращены как несколько значений. Например:

(values-list (list a b c)) ≡ (values a b c)

Можно обозначить так,

(values-list *list*) ≡ (apply #'values *list*)

но `values-list` может быть более ясным или эффективным.

[Макрос] **multiple-value-list** form

`multiple-value-list` вычисляет форму *form* и возвращает список из того количества значений, которое было возвращено формой. Например,

(multiple-value-list (floor -3 4)) ⇒ (-1 1)

Таким образом, `multiple-value-list` и `values-list` являются антиподами. FIXME

[Специальный оператор] **multiple-value-call** function {form}*

`multiple-value-call` сначала вычисляет *function* для получения функции и затем вычисляет все формы *forms*. Все значения форм *forms* собираются вместе (все, а не только первые) и передаются как аргументы функции. Результат `multiple-value-call` является тем, что вернула функция. Например:

(+ (floor 5 3) (floor 19 4))
  ≡ (+ 1 4) ⇒ 5
(multiple-value-call #'+ (floor 5 3) (floor 19 4))
  ≡ (+ 1 2 4 3) ⇒ 10
(multiple-value-list *form*) ≡ (multiple-value-call #'list *form*)

[Специальный оператор] **multiple-value-prog1** form {form}*

`multiple-value-prog1` вычисляет первую форму *form* и сохраняет все значения, возвращённые данной формой. Затем она слева направо вычисляет оставшиеся *формы*, игнорируя их значения. Значения, возвращённые первой формой, становятся результатом всей формы `multiple-value-prog1`. Смотрите `prog1`, которая всегда возвращает одно значение.

[Макрос] **multiple-value-bind** ({var}*) values-form
{declaration}* {form}*

Вычисляется *values-form* и каждое значение результата связывается соответственно с переменными указанными в первом параметре. Если переменных больше, чем возвращаемых значений, для оставшихся переменных используется значение `nil`. Если значений больше, чем переменных, лишние значения игнорируются. Переменные связываются со значениями только на время выполнения форм тела, которое является неявным `progn`. Например:

```
(multiple-value-bind (x) (floor 5 3) (list x)) ⇒ (1)
(multiple-value-bind (x y) (floor 5 3) (list x y)) ⇒ (1 2)
(multiple-value-bind (x y z) (floor 5 3) (list x y z))
  ⇒ (1 2 nil)
```

[Макрос] **multiple-value-setq** variables form

Аргумент *variables* должен быть списком переменных. Вычисляется форма *form* и переменным *присваиваются* (не связываются) значения, возвращённые этой формой. Если переменных больше, чем значений, оставшимся переменным присваивается `nil`. Если значений больше, чем переменных, лишние значения игнорируются.

`multiple-value-setq` всегда возвращает одно значение, которое является первым из возвращённых значений формы *form*, или `nil`, если форма *form* вернула ноль значений.

[Макрос] **nth-value** n form

Формы аргументов n и *form* вычисляются. Значение n должно быть неотрицательным целым, и форма *form* должна возвращать любое количество значение. Целое число n используется, как индекс (начиная с нуля), для доступа к списку значений. Форма возвращает элемент в позиции n из списка результатов вычисления формы *form*. Если позиции не существует, возвращается `nil`.

В качестве примера, `mod` мог бы быть определён так:

```
(defun mod (number divisor)
  (nth-value 1 (floor number divisor)))
```

7.10.2 Правила управления возвратом нескольких значений

Часто случается так, что значение специальной формы или макроса определено как значение одного из подвыражений. Например, значение `cond` является значением последнего подвыражения в исполняемой ветке.

В большинстве таких случаев, если подвыражение возвращает несколько значений, тогда и оригинальная форма возвращает все эти значения. Эта *передача значений*, конечно, не будет иметь место, если не была выполнена специальная форма для обработки возврата нескольких значений.

Неявно несколько значений может быть возвращены из специальных форм в соответствие со следующими правилами:

Вычисление и применение

- `eval` возвращает несколько значений, если переданная ему форму при вычислении вернула несколько значений.
- `apply`, `funcall` и `multiple-value-call` передают обратно несколько значений из применяемой или вызываемой функции.

В контексте неявного `progn`

- Специальная форма `progn` передаёт обратно несколько значений полученных при вычислении последней подформы. В другие ситуациях, называемых «неявный `progn`», в которых вычисляется несколько форм и результаты всех, кроме последней формы, игнорируются, также передаётся обратно несколько значений от формы. Такие ситуации включают тело лямбда-выражения, в частности в `defun`, `defmacro` и `deftype`. Также включаются тела конструкций `eval-when`, `progv`, `let`, `let*`, `when`, `unless`, `block`, `multiple-value-bind` и `catch`. И также включаются подвыражения условных конструкций `case`, `typecase`, `ecase`, `etypecase`, `ccase` и `ctypecase`.

Условные конструкции

- `if` передаёт обратно несколько значений из любой выбранной подформы (ветки *then* или *else*).
- `and` и `or` передают обратно несколько значений из последней подформы, но ни из какой другой непоследней подформы.
- `cond` передаёт обратно несколько значений из последней подформы неявного `progn` выделенного подвыражения. Однако, если выделенное подвыражение является «синглтоном», будет возвращено только одно значение (не-`nil` значение предиката). Это верно, даже если выражение «синглтон» является последним подвыражением `cond`. Нельзя рассматривать последнее подвыражение `(x)`, как будто оно `(t x)`. Последнее подвыражение передаёт обратно несколько значений из формы `x`.

Возврат из блока

- При нормальном завершении конструкция `block` передаёт обратно несколько значений из её последней подформы. Если для завершения использовалась `return-from` (или `return`), тогда `return-from` передаёт обратно несколько значений из своей подформы, как значения всей конструкции `block`. Другие конструкции, создающие неявные блоки, такие как `do`, `dolist`, `dotimes`, `prog` и `prog*`, также передают обратно несколько значений, заданных с помощью `return-from` (или `return`).
- `do` передаёт обратно несколько значений из последней формы подвыражения выхода, точно также, как если бы подвыражение выхода было подвыражение `cond`. Подобным образом действуют `dolist` и `dotimes`. Они возвращают несколько значений из формы *resultform*, если она была выполнена. Эти ситуации также являются примерами явного использования `return-from`.

Выход из ловушки исключения

- Конструкция `catch` возвращает несколько значений, если результат формы в `throw`, которая осуществляет выход из этого catch, возвращает несколько значений.

Остальные ситуации

- `multiple-value-prog1` передаёт обратно несколько значений из его первой подформы. Однако, `prog1` всегда возвращает одно значение.
- `unwind-protect` возвращает несколько значений, если форма, которую он защищает, вернула несколько значений.
- `the` возвращает несколько значений, если в нем содержащаяся форма возвращает несколько значений.

`prog1`, `prog2`, `setq` и `multiple-value-setq` — это формы, которые *никогда* не передают обратно несколько значений. Стандартный метод для явного указания возврата одного значения из формы `x` является запись `(values x)`.

Наиболее важное правило насчёт нескольких значений: **Не важно сколько значений возвращает форма, если форма является аргументом в вызове функции, то будет использовано только одно значение (первое).**

Например, если вы записали `(cons (floor x))`, тогда `cons` всегда получить *только* один аргумент (что, конечно, является ошибкой), хотя и `floor` возвращает два значения. Для того, чтобы поместить оба значения `floor` в `cons`, можно записать что-то вроде этого: `(multiple-value-call #'cons (floor x))`. В обычном вызове функции, каждый форма аргумента предоставляется только *один* аргумент. Если такая форма возвращает ноль значение, в качестве аргумента используется `nil`. А если возвращает более одного значения, все они, кроме первого, игнорируются. Также и условные конструкции, например `if`, которые проверяют значения формы, используют только одно первое значение, остальные игнорируются. Такие конструкции будут использовать `nil` если форма вернула ноль значений.

7.11 Динамические нелокальные выходы

Common Lisp предоставляет функциональность для выхода из сложного процесса в нелокальном, динамических ограниченном стиле. Для этих целей есть два вида специальных форм, называемых *catch* и *throw*. Форма catch выполняет некоторые подформы так, что если форма throw выполняется в ходе этих вычислений, в данной точке вычисления прерываются и форма catch немедленно возвращает значение указанное в throw. В отличие от `block` и `return` (раздел 7.7), которые позволяют выйти из тела `block` из любой точки лексически, находящейся внутри тела, `catch/throw` механизм работает, даже если форма `throw` по тексту не находится внутри тела формы `catch`. Throw может использовать только в течение (продолжительности времени) выполнения тела catch. Можно провести аналогию между ними, как между динамически связываемыми переменными и лексически связываемыми.

[Специальный оператор] **catch** tag {form}*

Специальная форма `catch` служит мишенью для передачи управления с помощью `throw`. Первой выполняется форма *tag* для создания объекта для имени catch. Он может быть любым Lisp'овым объектом. Затем устанавливается ловушка с тегом, в качестве которого используется этот объект. Формы *form* выполняются как неявный `progn`, и возвращается результат последней формы. Однако если во время вычислений будет выполнена форма `throw` с тегом, который равен `eq` тегу catch, то управление немедленно прерывается и возвращается результат указанный в форме throw. Ловушка, устанавливаемая с помощью `catch`, упраздняется перед тем, как будет возвращён результат.

Тег используется для соотнесения throws и catches. `(catch 'foo` *`form`*`)` будет перехватывать `(throw 'foo` *`form`*`)`, но не будет `(throw 'bar` *`form`*`)`. Если `throw` выполнен без соответствующего `catch`, готового его обработать, то это является ошибкой.

Теги catch сравниваются с использованием `eq`, а не `eql`. Таким образом числа и строковые символы не могут использоваться в качестве тегов.

[Специальный оператор] **unwind-protect** protected-form {cleanup-form}*

Иногда необходимо выполнить форму и быть уверенным, что

некоторые побочные эффекты выполняются после её завершения. Например:

```
(progn (start-motor)
       (drill-hole)
       (stop-motor))
```

Функциональность нелокальных выходов в Common Lisp создаёт ситуацию, в которой однако данный код может не сработать правильно. Если `drill-hole` может бросить исключение в ловушку, находящуюся выше данного `progn`, то `(stop-motor)` никогда не выполниться. Более удобный пример с открытием/закрытием файлов:

```
(prog2 (open-a-file)
       (process-file)
       (close-the-file))
```

где закрытие файла может не произойти, по причине ошибки в функции `process-file`.

Для того, чтобы вышеприведённый код работал корректно, можно переписать его с использованием `unwind-protect`:

```
;; Stop the motor no matter what (even if it failed to start).

(unwind-protect
  (progn (start-motor)
         (drill-hole))
  (stop-motor))
```

Если `drill-hole` бросит исключение, которое попытается выйти из блока `unwind-protect`, то `(stop-motor)` будет обязательно выполнена.

Этот пример допускает то, что вызов `stop-motor` корректен, даже если мотор ещё не был запущен. Помните, что ошибка или прерывание может осуществить выход перед тем, как будет проведена инициализация. Любой код по восстановлению состояния должен правильно работать вне зависимости от того, где произошла ошибка. Например, следующий код неправильный:

```
(unwind-protect
  (progn (incf *access-count*)
         (perform-access))
  (decf *access-count*))
```

Если выход случиться перед тем, как выполниться операция `incf`, то выполнение `decf` приведён к некорректному значению в `*access-count*`. Правильно будет записать этот код так:

```
(let ((old-count *access-count*))
  (unwind-protect
    (progn (incf *access-count*)
           (perform-access))
    (setq *access-count* old-count)))
```

Как правило, `unwind-protect` гарантирует выполнение форм *cleanup-form* перед выходом, как в случае нормального выхода, так и в случае генерации исключения. (Если, однако, выход случился в ходе выполнения форм *cleanup-form*, никакого специального действия не предпринимается. Формы *cleanup-form* не защищаются. Для этого необходимо использовать дополнительные конструкции `unwind-protect`.) `unwind-protect` возвращает результат выполнения защищённой формы *protected-form* и игнорирует все результаты выполнения форм чистки *cleanup-form*.

Следует подчеркнуть, что `unwind-protect` защищает против *всех* попыток выйти из защищённой формы, включая не только «динамический выход» с помощью `throw`, но и также «лексический выход» с помощью `go` и `return-from`. Рассмотрим следующую ситуацию:

```
(tagbody
  (let ((x 3))
    (unwind-protect
      (if (numberp x) (go out))
      (print x)))
 out
  ...)
```

Когда выполнится `go`, то сначала выполнится `print`, а затем перенос управления на тег `out` будет завершён.

[Специальный оператор] **throw** tag result

Специальная форма `throw` переносит управление к соответствующей конструкции `catch`. Сначала выполняется *tag* для вычисления объекта, называемого тег throw. Затем вычисляется форма результата *result*, и этот результат сохраняется (если форма *result* возвращает несколько значений, то *все* значения сохраняются). Управление выходит и самого последнего установленного catch, тег которого совпадает с тегом throw. Сохранённые результаты возвращаются, как значения конструкции `catch`. Теги catch и throw совпадают, только если они равны `eq`.

В процессе, связывания динамических переменных упраздняются до точки catch, и выполнятся все формы очистки в конструкциях unwind-protect. Форма *result* вычисляется перед началом процесса раскручивания, и её значение возвращается из блока catch.

Если внешний блок catch с совпадающим тегом не найден, раскрутка стека не происходит и сигнализируется ошибка. Когда ошибка сигнализируется, ловушки и динамические связывания переменных являются теми, которые были в точке throw.

Таблица 7.1: Structure Traversal Operations Subject to Side Effect Restrictions

adjoin	maphash	reduce
assoc	mapl	remove
assoc-if	maplist	remove-duplicates
assoc-if-not	member	remove-if
count	member-if	remove-if-not
count-if	member-if-not	search
count-if-not	merge	set-difference
delete	mismatch	set-exclusive-or
delete-duplicates	nintersection	some
delete-if	notany	sort
delete-if-not	notevery	stable-sort
do-all-symbols	nset-difference	sublis
do-external-symbols	nset-exclusive-or	subsetp
do-symbols	nsublis	subst
dolist	nsubst	subst-if
eval	nsubst-if	subst-if-not
every	nsubst-if-not	substitute
find	nsubstitute	substitute-if
find-if	nsubstitute-if	substitute-if-not
find-if-not	nsubstitute-if-not	tree-equal
intersection	nunion	union
certain loop clauses	position	with-hash-table-iterator
map	position-if	with-input-from-string
mapc	position-if-not	with-output-to-string
mapcan	rassoc	with-package-iterator
mapcar	rassoc-if	
mapcon	rassoc-if-not	

Глава 8

Макросы

Функциональность макросов Common Lisp'а позволяет пользователю определять произвольные функции, которые преобразуют заданные Lisp'овые формы в другие формы, перед тем, как они будут вычислены или скомпилированы. Это делается на уровне выражений, а не на уровне строк, как в других языках. При написании хорошего кода макросы важны: они предоставляют возможность писать код, которые ясен и элегантен на пользовательском уровне, но после преобразования становится сложным или более эффективным для выполнения.

Когда `eval` получает список, у которого *car* элемент является символом, она ищет локальные определения для этого символа (`flet`, `labels` и `macrolet`). Если поиски не увенчались успехом, она ищет глобальное определение. Если это глобальное определение является макросом, тогда исходный список называется *макровызовом*. С определением будет ассоциирована функция двух аргументов, называемая *функцией раскрытия*. Эта функция вызывается с макровызовом в качестве первого аргумента и лексическим окружением в качестве второго. Функция должна вернуть новую Lisp'овую форму, называемую *раскрытием* макровызова. (На самом деле участвует более общий механизм, смотрите `macroexpand`.) Затем это раскрытие выполняется по месту оригинальной (исходной) формы.

Когда функция компилируется, все макросы, в ней содержащиеся, раскрываются во время компиляции. Это значит, что определение макроса должно быть прочитано компилятором до его первого использования.

В целом, реализация Common Lisp'а имеет большую свободу в

выборе того, когда в программе раскрываются макровызовы. Например, допускается для специальной формы `defun` раскрытие всех внутренних макровызовов в время выполнения формы `defun` и записи полностью раскрытого тела функции, как определение данной функции для дальнейшего использования. (Реализация может даже выбрать путь, все время компилировать функции определённые с помощью `defun`, даже в режиме «интерпретации».)

Для правильного раскрытия макросы должны быть написаны так, чтобы иметь наименьшие зависимости от выполняемого окружения. Лучше всего удостовериться, что все определения макросов доступны перед тем, как компилятор или интерпретатор будет обрабатывает код, содержащий макровызовы к ним.

В Common Lisp, макросы не являются функциями. В частности, макросы не могут использоваться, как функциональные аргументы к таким функциям, как `apply`, `funcall` или `map`. В таких ситуациях список, отображающий «первоначальный макровызов» не существует и не может существовать, потому что в некотором смысле аргументы уже были вычислены.

8.1 Определение макроса

Функция `macro-function` определяет, является ли данный символ именем макроса. Конструкция `defmacro` предоставляет удобный способ определить новый макрос.

[Функция] **macro-function** *symbol* **&optional** *env*

Первый аргумент должен быть символом. Если символ содержит определение функции, то есть определение макроса, локально созданное в окружении *env* с помощью `macrolet` или глобально созданное с помощью `defmacro`, тогда возвращается функция раскрытия (функция двух аргументов, первый форма макровызова и второй — окружение). Если символ не содержит определение функции, или это определение обычной функции или это специальная форма, но не макрос, тогда возвращается `nil`. Наилучший способ вызвать функцию раскрытия использовать `macroexpand` или `macroexpand-1`.

Возможно такое, что и `macro-function`, и `special-operator-p` обе возвращают истину для одного заданного символа. Так происходит,

потому что реализация может для увеличения производительности реализовывать любой макрос, как специальную форму. С другой стороны, определения макросов должны быть доступны для использования программами, которые понимают только стандартные специальные формы, перечисленные в таблице 5.1.

`setf` может быть использована с `macro-function` для установки в символ глобального определения макроса:

(setf (macro-function *symbol*) *fn*)

Устанавливаемое значение должно быть функцией, которая принимает два аргумента, список макровызова и окружение, и вычислять раскрытие этого макровызова. Выполнение этой операции указывает на то, что символ будет содержать *только* это определение макроса в качестве определения глобальной функции. Предыдущее определение функции или макроса утрачивается. С помощью `setf` невозможно установить локальное определение макроса. При использовании `setf` указание второго параметра (окружение) является ошибкой. Переопределение специальной формы также является ошибкой.

Смотрите также `compiler-macro-function`.

[Макрос] **defmacro** name lambda-list [[{declaration}* | doc-string]] {form}*

`defmacro` является макросом определяющим макросы, которые преобразуют формы макровызовов. `defmacro` имеет почти такой же синтаксис, как и `defun`: *name* является символом, для которого создаётся макрос, *lambda-list* похож на список параметров лямбда-выражения и *form* содержит тело функции раскрытия. Конструкция *defmacro* устанавливает функцию раскрытия, как глобальное определение макроса для символа *name*. FIXME

Форма обычно используется на верхнем уровне, но её также можно использовать на неверхнем уровне. Таким образом, `defmacro` должна определять функцию раскрытия макроса в некотором лексическом, а не глобальном окружении.

Тело функции раскрытия, определяемое с помощью `defmacro`, неявно оборачивается в конструкцию `block`, имя которой совпадает с именем `name`, определяемого макроса. Таким образом для выхода из функции может использоваться `return-from`.

Форма `defmacro` возвращает в качестве значение *name*.

Если мы рассмотрим макровызов, как список содержащий имя функции и некоторые формы аргументов, то механизм заключается в передаче в функцию `apply` этой функции и списка её (невычисленных) аргументов. Спецификаторы параметров обрабатываются также, как и для любого лямбда-выражения. В качестве параметров используются формы аргументов макровызова. Затем вычисляются формы тела, как неявный `progn`. Значение последней формы возвращается, как раскрытие макровызова.

Если указана необязательная строка документации *doc-string*, тогда она присоединяется к символу *name*, как строка документации типа `function`. Смотрите `documentation`. Если в определении макроса представлена только строка документации, и после неё нет ни одной формы, ни деклараций, ни просто для тела, то данная строка сама становиться формой, и тело считается состоящим из одной формы — этой строки.

Следующие три дополнительных маркера доступны в определении лямбда-списка.

`&body` Данный маркер эквивалентен `&rest` маркеру, но дополнительно сообщает для функций вывода-форматирования и редактирования, что оставшаяся часть формы рассматривается как тело и должна быть правильно отформатирована. (Можно использовать исключительно один маркер или `&body`, или `&rest`)

`&whole` За данным маркером указывается одна переменная, которая будет связана со всей формой макровызова. Это значение, которое получает функция определения макроса в качестве своего первого аргумента. `&whole` и следующая переменная должны указываться на первой позиции в лямбда-списке, перед другими параметрами или маркерами.

`&environment` За данным маркером указывается одна переменная, которая будет связана с окружением, отображающим лексическое окружение, в котором произошёл макровызов. Это окружение может не быть полным лексическим окружением. Оно должно использоваться только с функцией `macroexpand` ради любых локальных определений макросов, которые могли быть установлены с помощью `macrolet` конструкции внутри этого лексического

окружения. Этот функционал полезен крайне редко, когда определение макроса должно явно раскрыть другие макросы в процессе своего раскрытия.

Смотрите `lambda-list-keywords`.

X3J13 voted in March 1989 to specify that macro environment objects received with the `&environment` argument of a macro function have only dynamic extent. The consequences are undefined if such objects are referred to outside the dynamic extent of that particular invocation of the macro function. This allows implementations to use somewhat more efficient techniques for representing environment objects.

X3J13 voted in March 1989 to clarify the permitted uses of `&body`, `&whole`, and `&environment`:

- `&body` may appear at any level of a `defmacro` lambda-list.

- `&whole` may appear at any level of a `defmacro` lambda-list. At inner levels a `&whole` variable is bound to that part of the argument that matches the sub-lambda-list in which `&whole` appears. No matter where `&whole` is used, other parameters or lambda-list keywords may follow it.

- `&environment` may occur only at the outermost level of a `defmacro` lambda-list, and it may occur at most once, but it may occur anywhere within that lambda-list, even before an occurrence of `&whole`.

`defmacro`, в отличие от любых других конструкций Common Lisp'a, имеющих лямбда-списки в своём синтаксисе, предоставляет дополнительную функциональность известную как *деструктуризация*.

Смотрите `destructuring-bind`, которая отдельно предоставляет эту функциональность.

В любом месте, где могло бы стоять имя параметра, и где по синтаксису не ожидается использование списка (как описано в разделе 5.2.2), можно использовать ещё один встроенный лямбда-список. Когда использован такой приём, при передаче форм аргументов для встроенного лямбда-списка, необходимо обернуть эти формы в отдельный список. В качестве примера, определение макроса для `dolist` можно записать в таком стиле:

```
(defmacro dolist ((var listform &optional resultform)
                  &rest body)
  ...)
```

Ниже будет больше примеров использования встраиваемых лямбда-списков в `defmacro`.

Следующим правилом деструктуризации является то, что `defmacro` позволяет любому лямбда-списку (верхнего уровня или встроенному) быть dotted, заканчивающимся именем параметра. Такая ситуация обрабатывается так, как будто имя параметра в конце списка, стоит после неявного маркера `&rest`. Например, уже показанное определение `dolist` может быть записано так:

```
(defmacro dolist ((var listform &optional resultform)
                  . body)
  ...)
```

Если компилятор встречает `defmacro`, он добавляет новый макрос в окружение компиляции, и также функция раскрытия добавляется в выходной файл. Таким образом новый макрос будет более быстрым во время выполнения. Если необходимо избежать такого механизма, можно использовать `defmacro` внутри конструкции `eval-when`.

`defmacro` может также использоваться для переопределения макроса (например, для установки корректной версии определения вместо некорректной), или для переопределения функции в макрос. Переопределение специальной формы (смотрите таблицу 5.1) не допускается. Смотрите `macrolet`, которая устанавливает определение макроса в замкнутом лексическом области видимости.

Допустим, для примера, что необходимо реализовать условную конструкцию аналогичную Fortran'овскому арифметическому выражению IF. (Это конечно требует определённого расширения воображения и приостановки неверия.) Конструкция должна принимать четыре формы: *test-value*, *neg-form*, *zero-form* и *pos-form*. В зависимости от того, является ли *test-form* отрицательным, нулём или положительным числом, для выполнения будет выбрана одна из трёх последних форм. С использованием `defmacro`, определение этой конструкции может выглядеть так:

```
(defmacro arithmetic-if (test neg-form zero-form pos-form)
  (let ((var (gensym)))
    `(let ((,var ,test))
       (cond ((< ,var 0) ,neg-form)
             ((= ,var 0) ,zero-form)
             (t ,pos-form)))))
```

Необходимо отметить, что в данном определении используется функциональность обратной кавычки (смотрите раздел 22.1.3). Также необходимо заметить, что используется `gensym` для создания нового имени переменной. Это необходимо для избежания конфликтов с другими переменными, которые могут использоваться в формах *neg-form*, *zero-form* или *pos-form*.

Если форма выполняется интерпретатором, то определение функции для символа `arithmetic-if` будет является макросом, с которым ассоциирована функция двух аргументом эквивалентная данной

```
(lambda (calling-form environment)
  (declare (ignore environment))
  (let ((var (gensym)))
    (list 'let
          (list (list 'var (cadr calling-form)))
          (list 'cond
                (list (list '< var '0) (caddr calling-form))
                (list (list '= var '0) (cadddr calling-form))
                (list 't (fifth calling-form))))))
```

Лямбда-выражение является результатом выполнения декларации `defmacro`. Вызов `list` является (гипотетически) результатом макросимвола обратной кавычки (`) и связанной с ним запятой. Конкретная функция раскрытия макроса может зависеть от реализации, например, она также может содержать проверку на корректность входных аргументов в макровызове.

Теперь, если `eval` встретит

```
(arithmetic-if (- x 4.0)
               (- x)
```

```
        (error "Strange zero")
        x)
```

то раскроет эту форму в

```
(let ((g407 (- x 4.0)))
  (cond ((< g407 0) (- x))
        ((= g407 0) (error "Strange zero"))
        (t x)))
```

и `eval` выполнит полученную форму. (Сейчас должно быть понятно, что функциональность обратной кавычки очень полезна для написания макросов. Она используется для построения шаблона, возвращаемой формы, с константными частями и частями для выполнения. Шаблон представляет собой «картину» кода, с местами для заполнения выделенными запятыми.)

Для улучшения примера мы можем сделать так, чтобы *pos-form* и *zero-form* могли быть заменены на `nil` в результате раскрытия макроса. Таким же образом действует и `if` опуская ветку *else* в случае истинности условия.

```
(defmacro arithmetic-if (test neg-form
                         &optional zero-form pos-form)
  (let ((var (gensym)))
    `(let ((,var ,test))
       (cond ((< ,var 0) ,neg-form)
             ((= ,var 0) ,zero-form)
             (t ,pos-form)))))
```

Тогда можно записать

```
(arithmetic-if (- x 4.0) (print x))
```

и это раскроется в что-то вроде

```
(let ((g408 (- x 4.0)))
  (cond ((< g408 0) (print x))
        ((= g408 0) nil)
        (t nil)))
```

Результирующий код корректен, но некрасиво выглядит. Можно переписать определение макроса для генерации лучшего кода, когда *pos-form* и *zero-form* могут быть вообще опущены. Или же можно положиться на реализацию Common Lisp'a, которая возможно оптимизирует этот код.

Деструктуризация является очень мощной функциональностью, которая позволяет лямбда-списку в `defmacro` выразить сложную структуру макровызова. Если не использовались ключевые символы лямбда-списка, то он представляет собой просто список с некоторой степенью вложенности и параметрами в качестве листьев. Структура макровызова должна иметь такую же структуру списка. Например, рассмотрим следующее определение макроса:

```
(defmacro halibut ((mouth eye1 eye2)
                   ((fin1 length1) (fin2 length2))
                   tail)
  ...)
```

Теперь давайте рассмотрим макровызов:

```
(halibut (m (car eyes) (cdr eyes))
         ((f1 (count-scales f1)) (f2 (count-scales f2)))
         my-favorite-tail)
```

Все это приведёт к тому, что функция раскрытия получит следующие значения для её параметров:

Параметр	Значение
mouth	m
eye1	(car eyes)
eye2	(cdr eyes)
fin1	f1
length1	(count-scales f1)
fin2	f2
length2	(count-scales f2)
tail	my-favorite-tail

Следующий макровызов ошибочный, так как аргумента для параметра `length1` не представлено:

```
(halibut (m (car eyes) (cdr eyes))
        ((f1) (f2 (count-scales f2)))
        my-favorite-tail)
```

Следующий макровызов также ошибочный, так как на месте предполагаемого списка указан символ.

```
(halibut my-favorite-head
        ((f1 (count-scales f1)) (f2 (count-scales f2)))
        my-favorite-tail)
```

Тот факт, что значение переменной `my-favorite-head` может быть списком, не имеет здесь значения. В макровызове структура должна совпадать с лямбда-списком в определении.

Использование ключевых символов лямбда-списка предоставляет ещё большую гибкость. Например, предположим, что удобно будет в функции раскрытия обращаться к элементам списка, называемым cd-mouth, `eye1`, and `eye2`, как к `head`. Можно записать так:

```
(defmacro halibut ((&whole head mouth eye1 eye2)
                   ((fin1 length1) (fin2 length2))
                   tail)
```

Теперь рассмотрим такой же, как раньше, корректный макровызов:

```
(halibut (m (car eyes) (cdr eyes))
        ((f1 (count-scales f1)) (f2 (count-scales f2)))
        my-favorite-tail)
```

Это приведёт к тому, что функции раскрытия получит те же значения для своих параметров, а также значение для параметра `head`:

Параметр	Значение
head	(m (car eyes) (cdr eyes))

Существует условие для деструктуризации, встроенный лямбда-список разрешён только на позиции, где синтаксис лямбда-списка предусматривает имя параметра, но не список. Это защищает от двусмысленности. Например, нельзя записать

```
(defmacro loser (x &optional (a b &rest c) &rest z)
  ...)
```

потому что синтаксис лямбда-списка не позволяет использовать списки после `&optional`. Список `(a b &rest c)` был бы интерпретирован как необязательный параметр `a`, у которого значение по умолчанию `b`, и supplied-p параметр с некорректным именем `&rest`, и дополнительным символом `c`, также некорректным. Было бы правильнее выразить это так:

```
(defmacro loser (x &optional ((a b &rest c)) &rest z)
  ...)
```

Дополнительные круглые скобки устраняют двусмысленность. Однако, такой макровызов, как `(loser (car pool))` не предоставляет никакой формы аргумента для лямбда-списка `(a b &rest c)`, значит значение по умолчанию для него будет `nil`. А так как `nil` является пустым списком, то этот макровызов ошибочен. Полностью корректное определение выглядит так:

```
(defmacro loser (x &optional ((a b &rest c) '(nil nil)) &rest z)
  ...)
```

или так

```
(defmacro loser (x &optional ((&optional a b &rest c)) &rest z)
  ...)
```

Они слегка отличаются: первое определение требует, что если макровызов явно указывает `a`, тогда он должен указать явно и `b`, тогда как второе определение не содержит такого требования. Например,

```
(loser (car pool) ((+ x 1)))
```

будет корректным макровызовом для второго определения, но не для первого.

8.2 Раскрытие макроса

Функция `macroexpand` служит для раскрытия макровызовов. Также предоставляется ловушка для пользовательской функции для управления процессом раскрытия.

[Функция] **macroexpand** *form* **&optional** *env*
[Функция] **macroexpand-1** *form* **&optional** *env*

Если форма *form* является макровызовом, тогда `macroexpand-1` раскроет макровызов только на *один* уровень и вернёт два значения: раскрытие и `t`. Если форма *form* не является макровызовом, тогда будут возвращены два значения: *form* и `nil`.

Форма *form* рассматривается как макровызов, только если она является cons-ячейкой, у которой *car* элемент является символом имени макроса. Окружение *env* похоже на то, что используется внутри вычислителя (смотрите `evalhook`). По-умолчанию равно нулевому окружению. Если окружение указано будут рассмотрены все установленные внутри *env* с помощью `macrolet` локальные определения макросов. Если в качестве аргумента указана только форма *form*, то берётся нулевое окружение, и будут рассматриваться только глобальные определения макросов (установленные с помощью `defmacro`).

Раскрытие макросов происходит следующим образом. Когда `macroexpand-1` устанавливает, что символ в форме указывает на макрос, тогда она получает функцию раскрытия для этого макроса. Затем вызывается значение переменной `*macroexpand-hook*`, как функция трёх аргументов с параметрами: функция раскрытия, форма *form* и окружение *env*. Значение, возвращённое этим вызовом, расценивается, как раскрытие макровызова. Значение переменной `*macroexpand-hook*` по-умолчанию `funcall`, которая следовательно просто запускает функцию раскрытия с двумя параметрами: формой *form* и окружением *env*.

X3J13 voted in June 1988 to specify that the value of `*macroexpand-hook*` is first coerced to a function before being called as the expansion interface hook. Therefore its value may be a symbol, a lambda-expression, or any object of type `function`.

X3J13 voted in March 1989 to specify that macro environment objects received by a `*macroexpand-hook*` function have only dynamic extent. The consequences are undefined if such objects are referred to outside the dynamic extent of that particular invocation of the hook function. This allows implementations to use somewhat more efficient techniques for representing environment objects.

X3J13 voted in June 1989 to clarify that, while `*macroexpand-hook*` may be useful for debugging purposes, despite the original design intent there is currently no correct portable way to use it for caching macro expansions.

- Caching by displacement (performing a side effect on the macro-call form) won't work because the same (`eq`) macro-call form may appear in distinct lexical contexts. In addition, the macro-call form may be a read-only constant (see `quote` and also section 24.1).

- Caching by table lookup won't work because such a table would have to be keyed by both the macro-call form and the environment, but X3J13 voted in March 1989 to permit macro environments to have only dynamic extent.

- Caching by storing macro-call forms and expansions within the environment object itself would work, but there are no portable primitives that would allow users to do this.

X3J13 also noted that, although there seems to be no correct portable way to use `*macroexpand-hook*` to cache macro expansions, there is no requirement

that an implementation call the macro expansion function more than once for a given form and lexical environment.

X3J13 voted in March 1989 to specify that `macroexpand-1` will also expand symbol macros defined by `symbol-macrolet`; therefore a *form* may also be a macro call if it is a symbol. The vote did not address the interaction of this feature with the `*macroexpand-hook*` function. An obvious implementation choice is that the hook function is indeed called and given a special expansion function that, when applied to the *form* (a symbol) and *env*, will produce the expansion, just as for an ordinary macro; but this is only my suggestion.

Вычислитель раскрывает макровызовы, как если бы использовал `macroexpand-1`. Таким образом `eval` также использует `*macroexpand-hook*`.

`macroexpand` похожа на `macroexpand-1`, однако циклично раскрывает форму *form*, пока в ней не останется макровызовов. (А точнее, `macroexpand` просто циклично вызывает `macroexpand-1`, пока последняя не вернёт `nil` во втором значении.) Второе возвращаемое значение функции `macroexpand-1` (`t` или `nil`) указывает на то, являлась ли форма *form* макровызовом.

[Переменная] ***macroexpand-hook***

Значение `*macroexpand-hook*` используется как интерфейс раскрытия для `macroexpand-1`.

8.3 Деструктуризация

[Макрос] **destructuring-bind** lambda-list expression {declaration}* {form}*

Данный макрос связывает переменные указанные в *лямбда-списке* с соответствующими значениями в древовидной структуре, являющейся результатом вычисления *выражения*, а затем выполняет *формы* как неявный `progn`.

Лямбда-список может содержать ключевые символы `&optional`, `&rest`, `&key` `&allow-other-keys`, and `&aux`. `&body` и `&whole` также могут использоваться как и в `defmacro`, однако `&environment` использоваться *не* может. Идея в том, что лямбда-список `destructuring-bind` имеет тот же формат, что и внутренние уровни лямбда-списка `defmacro`.

Если результат выполнения *выражения* не совпадает с шаблоном деструктуризации, то должна быть сигнализирована ошибка.

8.4 Compiler Macros

X3J13 voted in June 1989 to add a facility for defining *compiler macros* that take effect only when compiling code, not when interpreting it.

The purpose of this facility is to permit selective source-code transformations only when the compiler is processing the code. When the compiler is about to compile a non-atomic form, it first calls `compiler-macroexpand-1` repeatedly until there is no more expansion (there might not be any to begin with). Then it continues its remaining processing, which may include calling `macroexpand-1` and so on.

The compiler is required to expand compiler macros. It is unspecified whether the interpreter does so. The intention is that only the compiler will do so, but the range of possible "compiled-only" implementation strategies precludes any firm specification.

[Макрос] **define-compiler-macro** name lambda-list
{declaration | doc-string}* {form}*

This is just like `defmacro` except the definition is not stored in the symbol function cell of *name* and is not seen by `macroexpand-1`. It is, however, seen by `compiler-macroexpand-1`. As with `defmacro`, the *lambda-list* may include `&environment` and `&whole` and may include destructuring. The definition is global. (There is no provision for defining local compiler macros in the way that `macrolet` defines local macros.)

A top-level call to `define-compiler-macro` in a file being compiled by `compile-file` has an effect on the compilation environment similar to that of a call to `defmacro`, except it is noticed as a compiler macro (see section 24.1).

Note that compiler macro definitions do not appear in information returned by `function-information`; they are global, and their interaction with other lexical and global definitions can be reconstructed by `compiler-macro-function`. It is up to code-walking programs to decide whether to invoke compiler macro expansion.

X3J13 voted in March 1988 to specify that the body of the expander function defined by `defmacro` is implicitly enclosed in a `block` construct

whose name is the same as the *name* of the defined macro; presumably this applies also to `define-compiler-macro`. Therefore `return-from` may be used to exit from the function.

[Function] **compiler-macro-function** *name* **&optional** *env*

The *name* must be a symbol. If it has been defined as a compiler macro, then `compiler-macro-function` returns the macro expansion function; otherwise it returns `nil`. The lexical environment *env* may override any global definition for *name* by defining a local function or local macro (such as by `flet`, `labels`, or `macrolet`) in which case `nil` is returned.

`setf` may be used with `compiler-macro-function` to install a function as the expansion function for the compiler macro *name*, in the same manner as for `macro-function`. Storing the value `nil` removes any existing compiler macro definition. As with `macro-function`, a non-`nil` stored value must be a function of two arguments, the entire macro call and the environment. The second argument to `compiler-macro-function` must be omitted when it is used with `setf`.

[Function] **compiler-macroexpand** *form* **&optional** *env*
[Function] **compiler-macroexpand-1** *form* **&optional** *env*

These are just like `macroexpand` and `macroexpand-1` except that the expander function is obtained as if by a call to `compiler-macro-function` on the *car* of the *form* rather than by a call to `macro-function`. Note that `compiler-macroexpand` performs repeated expansion but `compiler-macroexpand-1` performs at most one expansion. Two values are returned, the expansion (or the original *form*) and a value that is true if any expansion occurred and `nil` otherwise.

There are three cases where no expansion happens:

- There is no compiler macro definition for the *car* of *form*.
- There is such a definition but there is also a `notinline` declaration, either globally or in the lexical environment *env*.
- A global compiler macro definition is shadowed by a local function or macro definition (such as by `flet`, `labels`, or `macrolet`).

Note that if there is no expansion, the original *form* is returned as the first value, and `nil` as the second value.

Any macro expansion performed by the function `compiler-macroexpand` or by the function `compiler-macroexpand-1` is carried out by calling the function that is the value of `*macroexpand-hook*`.

A compiler macro may decline to provide any expansion merely by returning the original form. This is useful when using the facility to put "compiler optimizers" on various function names. For example, here is a compiler macro that "optimizes" (one would hope) the zero-argument and one-argument cases of a function called `plus`:

```
(define-compiler-macro plus (&whole form &rest args)
  (case (length args)
    (0 0)
    (1 (car args))
    (t form)))
```

8.5 Environments

X3J13 voted in June 1989 to add some facilities for obtaining information from environment objects of the kind received as arguments by macro expansion functions, `*macroexpand-hook*` functions, and `*evalhook*` functions. There is a minimal set of accessors (`variable-information`, `function-information`, and `declaration-information`) and a constructor (`augment-environment`) for environments.

All of the standard declaration specifiers, with the exception of `special`, can be defined fairly easily using `define-declaration`. It also seems to be able to handle most extended declarations.

The function `parse-macro` is provided so that users don't have to write their own code to destructure macro arguments. This function is not entirely necessary since X3J13 voted in March 1989 to add `destructuring-bind` to the language. However, `parse-macro` is worth having anyway, since any program-analyzing program is going to need to define it, and the implementation isn't completely trivial even with `destructuring-bind` to build upon.

The function `enclose` allows expander functions to be defined in a non-null lexical environment, as required by the vote of X3J13 in March 1989 . It also provides a mechanism by which a program processing the body of an `(eval-when (:compile-toplevel) ...)` form can execute it in the enclosing environment (see issue).

In all of these functions the argument named *env* is an environment object. (It is not required that implementations provide a distinguished representation for such objects.) Optional *env* arguments default to `nil`, which represents the local null lexical environment (containing only global definitions and proclamations that are present in the run-time environment). All of these functions should signal an error of type `type-error` if the value of an environment argument is not a syntactic environment object.

The accessor functions `variable-information`, `function-information`, and `declaration-information` retrieve information about declarations that are in effect in the environment. Since implementations are permitted to ignore declarations (except for `special` declarations and `optimize safety` declarations if they ever compile unsafe code), these accessors are required only to return information about declarations that were explicitly added to the environment using `augment-environment`. They might also return information about declarations recognized and added to the environment by the interpreter or the compiler, but that is at the discretion of the implementor. Implementations are also permitted to canonicalize declarations, so the information returned by the accessors might not be identical to the information that was passed to `augment-environment`.

[Function] **variable-information** *variable* **&optional** *env*

This function returns information about the interpretation of the symbol *variable* when it appears as a variable within the lexical environment *env*. Three values are returned.

The first value indicates the type of definition or binding for *variable* in *env*:

`nil` There is no apparent definition or binding for *variable*.

`:special` The *variable* refers to a special variable, either declared or proclaimed.

`:lexical` The *variable* refers to a lexical variable.

`:symbol-macro` The *variable* refers to a `symbol-macrolet` binding.

`:constant` Either the *variable* refers to a named constant defined by `defconstant` or the *variable* is a keyword symbol.

The second value indicates whether there is a local binding of the name. If the name is locally bound, the second value is true; otherwise, the second value is `nil`.

The third value is an a-list containing information about declarations that apply to the apparent binding of the *variable*. The keys in the a-list are symbols that name declaration specifiers, and the format of the corresponding value in the *cdr* of each pair depends on the particular declaration name involved. The standard declaration names that might appear as keys in this a-list are:

`dynamic-extent` A non-`nil` value indicates that the *variable* has been declared `dynamic-extent`. If the value is `nil`, the pair might be omitted.

`ignore` A non-`nil` value indicates that the *variable* has been declared `ignore`. If the value is `nil`, the pair might be omitted.

`type` The value is a type specifier associated with the *variable* by a `type` declaration or an abbreviated declaration such as `(fixnum` *`variable`*`)`. If no explicit association exists, either by `proclaim` or `declare`, then the type specifier is `t`. It is permissible for implementations to use a type specifier that is equivalent to or a supertype of the one appearing in the original declaration. If the value is `t`, the pair might be omitted.

If an implementation supports additional declaration specifiers that apply to variable bindings, those declaration names might also appear in the a-list. However, the corresponding key must not be a symbol that is external in any package defined in the standard or that is otherwise accessible in the `common-lisp-user` package.

The a-list might contain multiple entries for a given key. The consequences of destructively modifying the list structure of this a-list or its elements (except for values that appear in the a-list as a result of `define-declaration`) are undefined.

Note that the global binding might differ from the local one and can be retrieved by calling `variable-information` with a null lexical environment.

[Function] **function-information** *function* **&optional** *env*

This function returns information about the interpretation of the

function-name *function* when it appears in a functional position within lexical environment *env*. Three values are returned.

The first value indicates the type of definition or binding of the function-name which is apparent in *env*:

`nil` There is no apparent definition for *function*.

`:function` The *function* refers to a function.

`:macro` The *function* refers to a macro.

`:special-form` The *function* refers to a special operator.

Some function-names can refer to both a global macro and a global special form. In such a case the macro takes precedence and `:macro` is returned as the first value.

The second value specifies whether the definition is local or global. If local, the second value is true; it is `nil` when the definition is global.

The third value is an a-list containing information about declarations that apply to the apparent binding of the function. The keys in the a-list are symbols that name declaration specifiers, and the format of the corresponding values in the *cdr* of each pair depends on the particular declaration name involved. The standard declaration names that might appear as keys in this a-list are:

`dynamic-extent` A non-`nil` value indicates that the function has been declared `dynamic-extent`. If the value is `nil`, the pair might be omitted.

`inline` The value is one of the symbols `inline`, `notinline`, or `nil` to indicate whether the function-name has been declared `inline`, declared `notinline`, or neither, respectively. If the value is `nil`, the pair might be omitted.

`ftype` The value is the type specifier associated with the function-name in the environment, or the symbol `function` if there is no functional type declaration or proclamation associated with the function-name. This value might not include all the apparent `ftype` declarations for the function-name. It is permissible for implementations to use a type specifier that is equivalent to or a supertype of the one that appeared in the original declaration. If the value is `function`, the pair might be omitted.

If an implementation supports additional declaration specifiers that apply to function bindings, those declaration names might also appear in the a-list. However, the corresponding key must not be a symbol that is external in any package defined in the standard or that is otherwise accessible in the `common-lisp-user` package.

The a-list might contain multiple entries for a given key. In this case the value associated with the first entry has precedence. The consequences of destructively modifying the list structure of this a-list or its elements (except for values that appear in the a-list as a result of `define-declaration`) are undefined.

Note that the global binding might differ from the local one and can be retrieved by calling `function-information` with a null lexical environment.

[Function] **declaration-information** *decl-name* **&optional** *env*

This function returns information about declarations named by the symbol *decl-name* that are in force in the environment *env*. Only declarations that do not apply to function or variable bindings can be accessed with this function. The format of the information that is returned depends on the *decl-name* involved.

It is required that this function recognize `optimize` and `declaration` as *decl-name*s. The values returned for these two cases are as follows:

`optimize` A single value is returned, a list whose entries are of the form (*`quality value`*), where *quality* is one of the standard optimization qualities (`speed`, `safety`, `compilation-speed`, `space`, `debug`) or some implementation-specific optimization quality, and *value* is an integer in the range 0 to 3 (inclusive). The returned list always contains an entry for each of the standard qualities and for each of the implementation-specific qualities. In the absence of any previous declarations, the associated values are implementation-dependent. The list might contain multiple entries for a quality, in which case the first such entry specifies the current value. The consequences of destructively modifying this list or its elements are undefined.

`declaration` A single value is returned, a list of the declaration names that have been proclaimed as valid through the use of the `declaration` proclamation. The consequences of destructively modifying this list or its elements are undefined.

If an implementation is extended to recognize additional declaration specifiers in `declare` or `proclaim`, it is required that either the `declaration-information` function should recognize those declarations also or the implementation should provide a similar accessor that is specialized for that declaration specifier. If `declaration-information` is used to return the information, the corresponding *decl-name* must not be a symbol that is external in any package defined in the standard or that is otherwise accessible in the `common-lisp-user` package.

[Function] **augment-environment** *env* **&key** *:variable :symbol-macro :function :macro :declare*

This function returns a new environment containing the information present in *env* augmented with the information provided by the keyword arguments. It is intended to be used by program analyzers that perform a code walk.

The arguments are supplied as follows.

`:variable` The argument is a list of symbols that will be visible as bound variables in the new environment. Whether each binding is to be interpreted as special or lexical depends on `special` declarations recorded in the environment or provided in the `:declare` argument.

`:symbol-macro` The argument is a list of symbol macro definitions, each of the form (*name definition*); that is, the argument is in the same format as the *cadr* of a `symbol-macrolet` special operator. The new environment will have local symbol-macro bindings of each symbol to the corresponding expansion, so that `macroexpand` will be able to expand them properly. A type declaration in the `:declare` argument that refers to a name in this list implicitly modifies the definition associated with the name. The effect is to wrap a `the` form mentioning the type around the definition.

`:function` The argument is a list of function-names that will be visible as local function bindings in the new environment.

`:macro` The argument is a list of local macro definitions, each of the form (*name definition*). Note that the argument is *not* in the same format as the *cadr* of a `macrolet` special operator. Each *definition*

must be a function of two arguments (a form and an environment). The new environment will have local macro bindings of each name to the corresponding expander function, which will be returned by `macro-function` and used by `macroexpand`.

`:declare` The argument is a list of declaration specifiers. Information about these declarations can be retrieved from the resulting environment using `variable-information`, `function-information`, and `declaration-information`.

The consequences of subsequently destructively modifying the list structure of any of the arguments to this function are undefined.

An error is signaled if any of the symbols naming a symbol macro in the `:symbol-macro` argument is also included in the `:variable` argument. An error is signaled if any symbol naming a symbol macro in the `:symbol-macro` argument is also included in a `special` declaration specifier in the `:declare` argument. An error is signaled if any symbol naming a macro in the `:macro` argument is also included in the `:function` argument. The condition type of each of these errors is `program-error`.

The extent of the returned environment is the same as the extent of the argument environment *env*. The result might share structure with *env* but *env* is not modified.

While an environment argument received by an `*evalhook*` function is permitted to be used as the environment argument to `augment-environment`, the consequences are undefined if an attempt is made to use the result of `augment-environment` as the environment argument for `evalhook`. The environment returned by `augment-environment` can be used only for syntactic analysis, that is, as an argument to the functions defined in this section and functions such as `macroexpand`.

[Макрос] **define-declaration** decl-name lambda-list {form}*

This macro defines a handler for the named declaration. It is the mechanism by which `augment-environment` is extended to support additional declaration specifiers. The function defined by this macro will be called with two arguments, a declaration specifier whose *car* is *decl-name* and the *env* argument to `augment-environment`. This function must return two values. The first value must be one of the following keywords:

`:variable` The declaration applies to variable bindings.

`:function` The declaration applies to function bindings.

`:declare` The declaration does not apply to bindings.

If the first value is `:variable` or `:function` then the second value must be a list, the elements of which are lists of the form (*binding-name key value*). If the corresponding information function (either `variable-information` or `function-information`) is applied to the *binding-name* and the augmented environment, the a-list returned by the information function as its third value will contain the *value* under the specified *key*.

If the first value is `:declare`, the second value must be a cons of the form (*key . value*). The function `declaration-information` will return *value* when applied to the *key* and the augmented environment.

`define-declaration` causes *decl-name* to be proclaimed to be a declaration; it is as if its expansion included a call (`proclaim '(declaration` *decl-name*)). As is the case with standard declaration specifiers, the evaluator and compiler are permitted, but not required, to add information about declaration specifiers defined with `define-declaration` to the macro expansion and `*evalhook*` environments.

The consequences are undefined if *decl-name* is a symbol that can appear as the *car* of any standard declaration specifier.

The consequences are also undefined if the return value from a declaration handler defined with `define-declaration` includes a *key* name that is used by the corresponding accessor to return information about any standard declaration specifier. (For example, if the first return value from the handler is `:variable`, the second return value may not use the symbols `dynamic-extent`, `ignore`, or `type` as *key* names.)

The `define-declaration` macro does not have any special compile-time side effects (see section 24.1).

[Function] **parse-macro** *name lambda-list body* **&optional** *env*

This function is used to process a macro definition in the same way as `defmacro` and `macrolet`. It returns a lambda-expression that accepts two arguments, a form and an environment. The *name*, *lambda-list*, and *body* arguments correspond to the parts of a `defmacro` or `macrolet` definition.

The *lambda-list* argument may include `&environment` and `&whole` and may include destructuring. The *name* argument is used to enclose the *body* in an implicit `block` and might also be used for implementation-dependent

purposes (such as including the name of the macro in error messages if the form does not match the *lambda-list*).

[Function] **enclose** *lambda-expression* **&optional** *env*

This function returns an object of type `function` that is equivalent to what would be obtained by evaluating `‘(function ,`*`lambda-expression`*`)` in a syntactic environment *env*. The *lambda-expression* is permitted to reference only the parts of the environment argument *env* that are relevant only to syntactic processing, specifically declarations and the definitions of macros and symbol macros. The consequences are undefined if the *lambda-expression* contains any references to variable or function bindings that are lexically visible in *env*, any `go` to a tag that is lexically visible in *env*, or any `return-from` mentioning a block name that is lexically visible in *env*.

Глава 9

Декларации

Декларации позволяют вам указать Lisp системе дополнительную информацию о вашей программе. С одним исключением, декларации абсолютно необязательные и корректность деклараций не означает корректность программы. Исключение из исключения в том, что декларации `special` *влияют* на интерпретацию связывания переменной и ссылки на неё, и *должны* указывается там, где необходимо. Все другие декларации носят рекомендательный характер, и могут использовать Lisp системой для создания дополнительных проверок ошибок или более производительного скомпилированного кода. Декларации также являются хорошим способом задокументировать программу.

Следует отметить, что нарушение декларации рассматривается, как ошибка (как, например, для декларации `type`), но реализация может не замечать этих ошибок (хотя их обнаружение где это возможно, поощряется).

9.1 Синтаксис декларации

Конструкция `declare` используется для встраивания деклараций внутрь выполняемого кода. Глобальные декларации и декларации, вычисленные программой, устанавливаются конструкцией `proclaim`.

Макрос `declaim`, который гарантированно распознаётся компилятором, и часто более удобен, чем `proclaim` для установки

глобальных деклараций. FIXME

[Специальный оператор] **declare** {decl-spec}*

Форма `declare` известна как *декларация*. Декларации могут использоваться только в начале тел соответствующих специальных форм. То есть декларация может использоваться в этой специальной форме, как выражение, и все предыдущие выражение (если есть) также должны быть формами `declare` (или, в некоторых случаях, строками документации). Декларации могут использоваться в лямбда-выражениях и перечисленных ниже формах.

```
define-setf-method      labels
defmacro                let
defsetf                 let*
deftype                 locally
defun                   macrolet
do                      multiple-value-bind
do*                     prog
do-all-symbols          prog*
do-external-symbols     with-input-from-string
do-symbols              with-open-file
dolist                  with-open-stream
dotimes                 with-output-to-string
flet                    with-conditions-restarts
print-unreadable-object with-standard-io-syntax
```

```
defgeneric              generic-function
define-method-combinatio generic-labels
defmethod               with-added-methods
generic-flet
```

```
symbol-macrolet         with-slots
with-accessors
```

Вычисление декларации является ошибкой. Специальные формы, которые позволяют использовать декларации, явно проверяют их наличие.

Макровызовы могут раскрываться в декларации, при условии, что макровызов указан в том месте, где могут быть указаны декларации. (Однако, макровызов не может использоваться в форме declare на месте *decl-spec*.)

Декларация может использовать только явно в теле соответсвтующего специального оператора в виде списка, *car* которого равен символу `declare`.

Каждая форма *decl-spec* является списком, у которого *car* элемент это символ, указывающий на тип декларации. Декларации могут быть разделены на два класса: одни относятся к связыванию переменных, другие нет. (Декларация `special` является исключением, она попадает в оба класса, это будет описано ниже.) Те, которые касаются связываний переменных, применяются только к связываниям, созданным в форме, в которой они используются. Например, в

```
(defun foo (x)
  (declare (type float x)) ...
  (let ((x 'a)) ...)
  ...)
```

декларация `type` применяется только для внешнего связывания `x`, а не для связывания, созданного в `let`.

Декларации, которые не воздействуют на связывания переменных, воздействуют на весь код тела в специальной форме. Например,

```
(defun foo (x y) (declare (notinline floor)) ...)
```

рекомендует, что везде внутри тела функции `foo`, `floor` должна быть вызвана как отдельная подпрограмма, а не встроена в код.

Некоторые специальные формы содержат части кода, которые, правильнее говоря, на являются телом этой формы. Это например код инициализации переменных и форма результата для циклов. Во всех случаях, такой дополнительный код оказывается под воздействием всеобъемлемых деклараций, которые указаны перед телом специальной

формы. Невсеобъемлемые декларации не оказывают воздействия на этот код, за исключением (конечно) ситуаций, когда код находится в области действия переменной, для которой использовалась декларация. Например:

```
(defun few (x &optional (y *print-circle*))
  (declare (special *print-circle*))
  ...)
```

Ссылка на `*print-circle*` в первой строке примера является специальной, так как указана соответствующая декларация.

```
(defun nonsense (k x z)
  (foo z x)              ;Первый вызов foo
  (let ((j (foo k x))    ;Второй вызов foo
        (x (* k k)))
    (declare (inline foo) (special x z))
    (foo x j z)))        ;Третий вызов foo
```

В этом примере, декларация `inline` применяется только ко второму и третьему вызову `foo`. Декларация `special` для переменной `x` указывает, что форма `let` создаст специальное связывание для `x` и тем самым ссылки в теле формы также будут специальными. Ссылка на `x` во втором вызове `foo` является специальной. Ссылка на `x` в первом вызове `foo` является локальной, а не специальной. Декларация `special` для `z` указывает на то, что ссылка в вызове `foo` будет специальной. Это значит ссылка не будет указывать на параметр функции `nonsense`, так как для параметра декларации `special` не указано. (Декларация `special` переменной `z` указывается не в теле `defun`, а в теле внутренней конструкции `let`. Таким образом она не воздействует на связывание параметра функции.) X3J13 voted in January 1989 to replace the rules concerning the scope of declarations occurring at the head of a special operator or lambda-expression:

- The scope of a declaration always includes the body forms, as well as any “stepper” or “result” forms (which are logically part of the body), of the special operator or lambda-expression.

- If the declaration applies to a name binding, then the scope of the declaration also includes the scope of the name binding.

Note that the distinction between pervasive and non-pervasive declarations is eliminated. An important change from the first edition is that "initialization" forms are specifically *not* included as part of the body under the first rule; on the other hand, in many cases initialization forms may fall within the scope of certain declarations under the second rule.

X3J13 also voted in January 1989 to change the interpretation of `type` declarations (see section 9.2).

These changes affect the interpretation of some of the examples from the first edition.

```
(defun foo (x)
  (declare (type float x)) ...
  (let ((x 'a)) ...)
  ...)
```

Under the interpretation approved by X3J13, the type declaration applies to *both* bindings of `x`. More accurately, the type declaration is considered to apply to variable references rather than bindings, and the type declaration refers to every reference in the body of `foo` to a variable named `x`, no matter to what binding it may refer.

```
(defun foo (x y) (declare (notinline floor)) ...)
```

This example of the use of `notinline` stands unchanged, but the following slight extension of it would change:

```
(defun foo (x &optional (y (floor x)))
  (declare (notinline floor)) ...)
```

Under first edition rules, the `notinline` declaration would be considered to apply to the call to `floor` in the initialization form for `y`. Under the interpretation approved by X3J13, the `notinline` would *not* apply to that particular call to `floor`. Instead the user must write something like

```
(defun foo (x &optional (y (locally (declare (notinline floor))
                                    (floor x))))
  (declare (notinline floor)) ...)
```

or perhaps

```
(locally (declare (notinline floor))
  (defun foo (x &optional (y (floor x))) ...))
```

Similarly, the `special` declaration in

```
(defun few (x &optional (y *print-circle*))
  (declare (special *print-circle*))
  ...)
```

is not considered to apply to the reference in the initialization form for `y` in `few`. As for the `nonsense` example,

```
(defun nonsense (k x z)
  (foo z x)              ;First call to foo
  (let ((j (foo k x))    ;Second call to foo
        (x (* k k)))
    (declare (inline foo) (special x z))
    (foo x j z)))        ;Third call to foo
```

under the interpretation approved by X3J13, the `inline` declaration is no longer considered to apply to the second call to `foo`, because it is in an initialization form, which is no longer considered in the scope of the declaration. Similarly, the reference to `x` in that second call to `foo` is no longer taken to be a special reference, but a local reference to the second parameter of `nonsense`.

`locally` выполняет формы *form* как неявный `progn` и возвращает одно или несколько значений последней формы.

[Специальный оператор] **locally** {declaration}* {form}*

Когда оператор `locally` используется на верхнем уровне, тогда формы в его теле выполняются как формы верхнего уровня. Это

означает что, например, `locally` можно использовать для оборачивания деклараций вокруг форм `defun` или `defmacro`.

```
(locally
  (declare (optimize (safety 3) (space 3) (debug 3) (speed 1)))
  (defun foo (x &optional (y (abs x)) (z (sqrt y)))
    (bar x y z)))
```

Без уверенности, что это работает, можно записать что-то вроде этого:

```
(defun foo (x &optional (y (locally
                             (declare (optimize (safety 3)
                                                (space 3)
                                                (debug 3)
                                                (speed 1)))
                             (abs x)))
                        (z (locally
                             (declare (optimize (safety 3)
                                                (space 3)
                                                (debug 3)
                                                (speed 1)))
                             (sqrt y))))
  (locally
    (declare (optimize (safety 3) (space 3) (debug 3) (speed 1)))
    (bar x y z)))
```

[Функция] **proclaim** *decl-spec*

Функция `proclaim` в качестве аргумента принимает *decl-spec*, и применяет указания в глобальном пространстве. (Такие глобальные декларации называются *прокламации.*) Так как `proclaim` является функцией, то её аргумент вычисляется всегда. Это позволяет программам вычислять декларацию и затем для применения помещать её в вызов `proclaim`.

Любые упоминаемые имена переменных указывают на значения динамических переменных. Например, выполненная прокламация

(proclaim '(type float tolerance))

указывает на то, что динамическое значение `tolerance` должно быть всегда числом с плавающей точкой. Подобным образом, любые упоминаемые имена функций указывают на глобальные определения функций.

Прокламации содержат универсальные декларации, которые всегда действуют, кроме случаев сокрытия их локальными декларациями. Например,

(proclaim '(inline floor))

рекомендует то, что `floor` должна быть встроена в места её вызова. Но в ситуации

(defun foo (x y) (declare (notinline floor)) ...)

`floor` будет вызываться как отдельная функция, так как локальная декларация скрыла прокламацию.

X3J13 voted in January 1989 to clarify that such shadowing does not occur in the case of type declarations. If there is a local type declaration for a special variable and there is also a global proclamation for that same variable, then the value of the variable within the scope of the local declaration must be a member of the intersection of the two declared types. This is consistent with the treatment of nested local type declarations on which X3J13 also voted in January 1989 .

Специальный случай, когда `proclaim` обрабатывает `special`, то *decl-spec* применяется ко всем связываниям и ссылкам на упомянутую переменную. Например, после

(proclaim '(special x))

в определении функции

(defun example (x) ...)

параметр `x` будет связан, как специальная (динамическая) переменная, а не лексическая (статическая). Такой приём должен использоваться аккуратно. Обычный способ определить глобальную специальную переменную это использовать `defvar` или `defparameter`.

[Макрос] **declaim** {decl-spec}*

Этот макрос синтаксически похож на `declare` и семантически на `proclaim`. Это выполняемая форма и она может использоваться везде, где может `proclaim`. Однако, формы *decl-spec* не вычисляются.

Если вызов этого макроса произошёл на верхнем уровне в файле, обрабатываемом компилятором, то прокламации также будут выполнены во время компиляции. As with other defining macros, it is unspecified whether or not the compile-time side effects of a `declaim` persist after the file has been compiled (see section 24.1). FIXME

9.2 Спецификаторы деклараций

Ниже представлен список спецификаторов деклараций для использования в `declare`.

`special (special` *`var1 var2 ...`*`)` указывает на то, что все указанные переменные должны рассматриваться как *специальные*. Данный спецификатор воздействует как на связывания переменных, так и на ссылки на эти переменные в коде. Все указанные связывания переменных будут динамическими, и ссылки будут осуществляться на эти связывания, а не на связывания определённые локально. Например:

```
(defun hack (thing *mod*)       ;Связывание параметра
  (declare (special *mod*))     ; *mod* доступно для hack1,
  (hack1 (car thing)))          ; но связывание thing нет

(defun hack1 (arg)
  (declare (special *mod*))     ;Декларируем что ссылка на *mod*
                                ; внутри hack1 будет специальной
  (if (atom arg) *mod*
```

```
(cons (hack1 (car arg)) (hack1 (cdr arg)))))
```

Следует отметить, что по правилам хорошего тона, имена специальных переменных окружаются звёздочками.

Декларация `special` *не* охватывает все связывания. Внутренние связывания переменных неявно скрывают декларацию `special` и должны быть явно передекларированы в специальные. (Однако прокламация `special` имеет глобальный эффект. Это сделано для совместимости с MacLisp'ом.) Например:

```
(proclaim '(special x))     ;x всегда специальная

(defun example (x y)
  (declare (special y))
  (let ((y 3) (x (* x 2)))
    (print (+ y (locally (declare (special y)) y)))
    (let ((y 4)) (declare (special y)) (foo x))))
```

В приведенном выше коде, внешние и внутренние связывания переменной `y` являются специальными и, таким образом, действуют в динамической области видимости. Однако среднее связывание действует в лексической области видимости. Два аргумента `+` различаются, один содержит значение `3` лексически связанной переменной `y`, другой содержит значение специальной переменной `y`. Все связывания и переменной `x` и ссылки на неё является специальными, так как в коде использовалась прокламация `special`.

В целях стиля, необходимо избегать использование прокламаций `special`. Для объявления специальных переменных предназначены макросы `defvar` и `defparameter`.

`type` `(type` *`type var1 var2 ...`*`)` применяются только для связываний переменный и указывает на то, что переменные будут принимать значения только указанного типа. В частности, значения присваиваемые переменным с помощью `setq`, как и первоначальные значения, должны быть заданного типа.

X3J13 voted in January 1989 to alter the interpretation of type declarations. They are not to be construed to affect "only variable bindings." The new rule for a declaration of a variable to have a specified type is threefold:

- It is an error if, during the execution of any reference to that variable within the scope of the declaration, the value of the variable is not of the declared type.
- It is an error if, during the execution of a `setq` of that variable within the scope of the declaration, the new value for the variable is not of the declared type.
- It is an error if, at any moment that execution enters the scope of the declaration, the value of the variable is not of the declared type.

One may think of a type declaration `(declare (type face bodoni))` as implicitly changing every reference to `bodoni` within the scope of the declaration to `(the face bodoni)`; changing every expression *exp* assigned to `bodoni` within the scope of the declaration to `(the face` *exp* `)`; and implicitly executing `(the face bodoni)` every time execution enters the scope of the declaration.

These new rules make type declarations much more useful. Under first edition rules, a type declaration was useless if not associated with a variable binding; declarations such as in

```
(locally
  (declare (type (byte 8) x y))
  (+ x y))
```

at best had no effect and at worst were erroneous, depending on one's interpretation of the first edition. Under the interpretation approved by X3J13, such declarations have "the obvious natural interpretation."

X3J13 noted that if nested type declarations refer to the same variable, then all of them have effect; the value of the variable must be a member of the intersection of the declared types.

Nested type declarations could occur as a result of either macro expansion or carefully crafted code. There are three cases. First, the inner type might be a subtype of the outer one:

```
(defun compare (apples oranges)
  (declare (type number apples oranges))
  (cond ((typep apples 'fixnum)
         ;; The programmer happens to know that, thanks to
         ;; constraints imposed by the caller, if APPLES
         ;; is a fixnum, then ORANGES will be also, and
         ;; therefore wishes to avoid the unnecessary cost
         ;; of checking ORANGES.  Nevertheless the compiler
         ;; should be informed to allow it to optimize code.
         (locally (declare (type fixnum apples oranges)))
                  ;; Maybe the compiler could have figured
                  ;; out by flow analysis that APPLES must
                  ;; be a fixnum here, but it doesn't hurt
                  ;; to say it explicitly.
           (< apples oranges)))
        ((or (complex apples)
             (complex oranges))
         (error "Not yet implemented.  Sorry."))
        ...))
```

This is the case most likely to arise in code written completely by hand.

Second, the outer type might be a subtype of the inner one. In this case the inner declaration has no additional practical effect, but it is harmless. This is likely to occur if code declares a variable to be of a very specific type and then passes it to a macro that then declares it to be of a less specific type.

Third, the inner and outer declarations might be for types that overlap, neither being a subtype of the other. This is likely to occur only as a result of macro expansion. For example, user code might declare a variable to be of type `integer`, and a macro might later declare it to be of type `(or fixnum package)`; in this case a compiler could intersect

the two types to determine that in this instance the variable may hold only fixnums.

The reader should note that the following code fragment is, perhaps astonishingly, *not in error* under the interpretation approved by X3J13:

```
(let ((james .007)
      (maxwell 86))
  (flet ((spy-swap ()
           (rotatef james maxwell)))
    (locally (declare (integer maxwell))
      (spy-swap)
      (view-movie "The Sound of Music")
      (spy-swap)
      maxwell)))
⇒ 86  (after a couple of hours of Julie Andrews)
```

The variable `maxwell` is declared to be an integer over the *scope* of the type declaration, not over its *extent*. Indeed `maxwell` takes on the non-integer value `.007` while the Trapp family make their escape, but because no reference to `maxwell` within the scope of the declaration ever produces a non-integer value, the code is correct.

Now the assignment to `maxwell` during the first call to `spy-swap`, and the reference to `maxwell` during the second call, *do* involve non-integer values, but they occur within the body of `spy-swap`, which is *not* in the scope of the type declaration! One could put the declaration in a different place so as to include `spy-swap` in the scope:

```
(let ((james .007)
      (maxwell 86))
  (locally (declare (integer maxwell))
    (flet ((spy-swap ()
             (rotatef james maxwell)))
      (spy-swap)                          ;Bug!
      (view-movie "The Sound of Music")
      (spy-swap)
      maxwell)))
```

and then the code is indeed in error.

X3J13 also voted in January 1989 to alter the meaning of the `function` type specifier when used in `type` declarations (see section 4.5).

type (*type var1 var2 ...*) является аббревиатурой для (type *type var1 var2 ...*), при условии, что *type* один из символов из таблицы 4.1.

Observe that this covers the particularly common case of declaring numeric variables:

```
(declare (single-float mass dx dy dz)
         (double-float acceleration sum))
```

In many implementations there is also some advantage to declaring variables to have certain specialized vector types such as `base-string`.

`ftype` (ftype *type function-name-1 function-name-2 ...*)
указывает но то, что именованная функция будет принимать и возвращать заданный типы, как в следующем примере:

```
(declare (ftype (function (integer list) t) nth)
         (ftype (function (number) float) sin cos))
```

Следует отметить, что соблюдаются правила лексической области видимости. Если одна из указанных функций лексически ограничена (с помощью `flet` или `labels`), то декларация применится для локального определения, а не для глобального.

X3J13 voted in March 1989 to extend `ftype` declaration specifiers to accept any function-name (a symbol or a list whose *car* is `setf`—see section 7.1). Thus one may write

```
(declaim (ftype (function (list) t) (setf cadr)))
```

to indicate the type of the `setf` expansion function for `cadr`.

X3J13 voted in January 1989 to alter the meaning of the `function` type specifier when used in `ftype` declarations (see section 4.5).

X3J13 voted in January 1989 to remove this interpretation of the `function` declaration specifier from the language. Instead, a declaration specifier

(function *var1 var2* ...)

is to be treated simply as an abbreviation for

(type function *var1 var2* ...)

just as for all other symbols appearing in table 4.1.

X3J13 noted that although `function` appears in table 4.1, the first edition also discussed it explicitly, with a different meaning, without noting whether the differing interpretation was to replace or augment the interpretation regarding table 4.1. Unfortunately there is an ambiguous case: the declaration

(declare (function foo nil string))

can be construed to abbreviate either

(declare (ftype (function () string) foo))

or

(declare (type function foo nil string))

The latter could perhaps be rejected on semantic grounds: it would be an error to declare `nil`, a constant, to be of type `function`. In any case, X3J13 determined that the ice was too thin here; the possibility of confusion is not worth the convenience of an abbreviation for `ftype` declarations. The change also makes the language more consistent.

`inline` (`inline` *`function1 function2 ...`*) указывает, что *было бы неплохо* встроить код указанных функций непосредственно в место вызова. Это может ускорить работу программы, но затруднить отладку (например, вызовы встроенных функций не могут быть протрассированы). Помните, компилятор может игнорировать эту декларацию.

Следует отметить, что соблюдаются правила лексической области видимости. Если одна из указанных функций лексически ограничена (с помощью `flet` или `labels`), то декларация применится для локального определения, а не для глобального.

X3J13 voted in October 1988 to clarify that during compilation the `inline` declaration specifier serves two distinct purposes: it indicates not only that affected calls to the specified functions should be expanded in-line, but also that affected definitions of the specified functions must be recorded for possible use in performing such expansions.

Looking at it the other way, the compiler is not required to save function definitions against the possibility of future expansions unless the functions have already been proclaimed to be `inline`. If a function is proclaimed (or declaimed) `inline` before some call to that function but the current definition of that function was established before the proclamation was processed, it is implementation-dependent whether that call will be expanded in-line. (Of course, it is implementation-dependent anyway, because a compiler is always free to ignore `inline` declaration specifiers. However, the intent of the committee is clear: for best results, the user is advised to put any `inline` proclamation of a function before any definition of or call to that function.)

Consider these examples:

```
(defun huey (x) (+ x 100))          ;Compiler need not remember this
(declaim (inline huey dewey))
(defun dewey (y) (huey (sqrt y)))  ;Call to huey unlikely to be expanded
(defun louie (z) (dewey (/ z)))    ;Call to dewey likely to be expanded
```

X3J13 voted in March 1989 to extend `inline` declaration specifiers to accept any function-name (a symbol or a list whose *car* is `setf`—see section 7.1). Thus one may write `(declare (inline (setf cadr)))` to indicate that the `setf` expansion function for `cadr` should be compiled in-line.

`notinline` `(notinline` *`function1 function2 ...`*`)` указывает на то, что встраивать код указанных функций в место вызова *не нужно*. Эта декларация обязательна. Компилятор *не* может её игнорировать.

Следует отметить, что соблюдаются правила лексической области видимости. Если одна из указанных функций лексически ограничена (с помощью `flet` или `labels`), то декларация применится для локального определения, а не для глобального. X3J13 voted in March 1989 to extend `notinline` declaration specifiers to accept any function-name (a symbol or a list whose *car* is `setf`—see section 7.1). Thus one may write `(declare (notinline (setf cadr)))` to indicate that the `setf` expansion function for `cadr` should not be compiled in-line.

X3J13 voted in January 1989 to clarify that the proper way to define a function `gnards` that is not `inline` by default, but for which a local declaration `(declare (inline gnards))` has half a chance of actually compiling `gnards` in-line, is as follows:

```
(declaim (inline gnards))

(defun gnards ...)

(declaim (notinline gnards))
```

The point is that the first declamation informs the compiler that the definition of `gnards` may be needed later for in-line expansion, and the second declamation prevents any expansions unless and until it is overridden.

While an implementation is never required to perform in-line expansion, many implementations that do support such expansion will not process `inline` requests successfully unless definitions are written with these proclamations in the manner shown above.

`ignore (ignore` *`var1 var2 ... varn`*`)` применяется только для связываний переменных и указывает на то, что связывания указанных переменных никогда не используются. Желательно, чтобы компилятор выдавал предупреждение, если переменная не используется и не задекларирована игнорироваться.

`optimize (optimize (`*`quality1 value1`*`) (`*`quality2 value2`*`)...)` рекомендует компилятору использовать указанные значения для свойств компилятора. Свойство компилятора является символом. Стандартные свойства включают `speed` (скомпилированного кода), `space` (и размер кода и run-time space), `safety` (проверка ошибок во время выполнения) и `compilation-speed` (скорость процесса компиляции).

X3J13 voted in October 1988 to add the standard quality `debug` (ease of debugging).

Также реализацией могут представляться другие свойства. *value* должно быть неотрицательным целым числом, обычно на интервале `0` и `3`. Значение `0` означает, что свойство полностью не важно, и `3`, что свойство полностью важно. `1` и `2` промежуточные значение, `1` «нормальный» или «обычный» уровень. Для наизначимого уровня существует аббревиатура, то есть `(`*`quality`* `3)` можно записать, как просто *quality*. Например:

```
(defun often-used-subroutine (x y)
  (declare (optimize (safety 2)))
  (error-check x y)
  (hairy-setup x)
```

```
(do ((i 0 (+ i 1))
     (z x (cdr z)))
    ((null z) i)
  ;; This inner loop really needs to burn.
  (declare (optimize speed))
  (declare (fixnum i))
  )))
```

`declaration (declaration` *name1 name2 ...*`)` сообщается компилятору, что каждая *namej* является нестандартным именем декларации. Целью является указать одному компилятор не создавать предупреждений для деклараций, которые используются для другого компилятора.

The `declaration` declaration specifier may be used with `declaim` as well as `proclaim`. The preceding examples would be better written using `declaim`, to ensure that the compiler will process them properly.

```
(declaim (declaration author
                      target-language
                      target-machine))

(declaim (target-language ada)
         (target-machine IBM-650))

(defun strangep (x)
  (declare (author "Harry Tweeker"))
  (member x '(strange weird odd peculiar)))
```

`dynamic-extent (dynamic-extent` *item1 item2 ... itemn*`)` declares that certain variables or function-names refer to data objects whose extents may be regarded as dynamic; that is, the declaration may be construed as a guarantee on the part of the programmer that the program will behave correctly even if the data objects have only dynamic extent rather than the usual indefinite extent.

Each *item* may be either a variable name or `(function f)` where f is a function-name (see section 7.1). (Of course, `(function f)` may be abbreviated in the usual way as `#'f`.)

It is permissible for an implementation simply to ignore this declaration. In implementations that do not ignore it, the compiler (or interpreter) is free to make whatever optimizations are appropriate given this information; the most common optimization is to stack-allocate the initial value of the object. The data types that can be optimized in this manner may vary from implementation to implementation.

The meaning of this declaration can be stated more precisely. We say that object x is an *otherwise inaccessible part* of y if and only if making y inaccessible would make x inaccessible. (Note that every object is an otherwise inaccessible part of itself.) Now suppose that construct c contains a `dynamic-extent` declaration for variable (or function) v (which need not be bound by c). Consider the values $w_1, \ldots, w_n$ taken on by v during the course of some execution of c. The declaration asserts that if some object x is an otherwise inaccessible part of w_j whenever w_j becomes the value of v, then just after execution of c terminates x will be either inaccessible or still an otherwise inaccessible part of the value of v. If this assertion is ever violated, the consequences are undefined.

In some implementations, it is possible to allocate data structures in a way that will make them easier to reclaim than by general-purpose garbage collection (for example, on the stack or in some temporary area). The `dynamic-extent` declaration is designed to give the implementation the information necessary to exploit such techniques.

For example, in the code fragment

```
(let ((x (list 'a1 'b1 'c1))
      (y (cons 'a2 (cons 'b2 (cons 'c2 'd2)))))
  (declare (dynamic-extent x y))
  ...)
```

it is not difficult to prove that the otherwise inaccessible parts of `x` include the three conses constructed by `list`, and that the otherwise

inaccessible parts of `y` include three other conses manufactured by the three calls to `cons`. Given the presence of the `dynamic-extent` declaration, a compiler would be justified in stack-allocating these six conses and reclaiming their storage on exit from the `let` form.

Since stack allocation of the initial value entails knowing at the object's creation time that the object can be stack-allocated, it is not generally useful to declare `dynamic-extent` for variables that have no lexically apparent initial value. For example,

```
(defun f ()
  (let ((x (list 1 2 3)))
    (declare (dynamic-extent x))
    ...))
```

would permit a compiler to stack-allocate the list in `x`. However,

```
(defun g (x) (declare (dynamic-extent x)) ...)
(defun f () (g (list 1 2 3)))
```

could not typically permit a similar optimization in `f` because of the possibility of later redefinition of `g`. Only an implementation careful enough to recompile `f` if the definition of `g` were to change incompatibly could stack-allocate the list argument to `g` in `f`.

Other interesting cases are

```
(declaim (inline g))
(defun g (x) (declare (dynamic-extent x)) ...)
(defun f () (g (list 1 2 3)))
```

and

```
(defun f ()
  (flet ((g (x) (declare (dynamic-extent x)) ...))
    (g (list 1 2 3))))
```

In each case some compilers might realize the optimization is possible and others might not.

An interesting variant of this is the so-called *stack-allocated rest list*, which can be achieved (in implementations supporting the optimization) by

```
(defun f (&rest x)
  (declare (dynamic-extent x))
  ...)
```

Note here that although the initial value of `x` is not explicitly present, nevertheless in the usual implementation strategy the function `f` is responsible for assembling the list for `x` from the passed arguments, so the `f` function can be optimized by a compiler to construct a stack-allocated list instead of a heap-allocated list.

Some Common Lisp functions take other functions as arguments; frequently the argument function is a so-called *downward funarg*, that is, a functional argument that is passed only downward and whose extent may therefore be dynamic.

```
(flet ((gd (x) (atan (sinh x))))
  (declare (dynamic-extent #'gd))     ;mapcar won't hang on to gd
  (mapcar #'gd my-list-of-numbers))
```

The following three examples are in error, since in each case the value of `x` is used outside of its extent.

```
(length (let ((x (list 1 2 3)))
          (declare (dynamic-extent x))
          x))                                ;Wrong
```

The preceding code is obviously incorrect, because the cons cells making up the list in `x` might be deallocated (thanks to the declaration) before `length` is called.

```
(length (list (let ((x (list 1 2 3)))
                (declare (dynamic-extent x))
                x)))                          ;Wrong
```

In this second case it is less obvious that the code is incorrect, because one might argue that the cons cells making up the list in `x` have no effect on the result to be computed by `length`. Nevertheless the code briefly violates the assertion implied by the declaration and is therefore incorrect. (It is not difficult to imagine a perfectly sensible implementation of a garbage collector that might become confused by a cons cell containing a dangling pointer to a list that was once stack-allocated but then deallocated.)

```
(progn (let ((x (list 1 2 3)))
         (declare (dynamic-extent x))
         x)                                   ;Wrong
       (print "Six dollars is your change have a nice day NEXT!"))
```

In this third case it is even less obvious that the code is incorrect, because the value of `x` returned from the `let` construct is discarded right away by the `progn`. Indeed it is, but "right away" isn't fast enough. The code briefly violates the assertion implied by the declaration and is therefore incorrect. (If the code is being interpreted, the interpreter might hang on to the value returned by the `let` for some time before it is eventually discarded.)

Here is one last example, one that has little practical import but is theoretically quite instructive.

```
(dotimes (j 10)
  (declare (dynamic-extent j))
  (setq foo 3)                ;Correct
  (setq foo j))               ;Erroneous—but why? (see text)
```

Since `j` is an integer by the definition of `dotimes`, but `eq` and `eql` are not necessarily equivalent for integers, what are the otherwise inaccessible parts of `j`, which this declaration requires the body of the `dotimes` not to "save"? If the value of `j` is `3`, and the body does `(setq foo 3)`, is that an error? The answer is no, but the interesting thing is that it depends on the implementation-dependent behavior of `eq` on numbers. In an implementation where `eq` and `eql` are equivalent for `3`, then `3` is not an otherwise inaccessible part because `(eq j (+ 2 1))` is true, and therefore there is another way to access the object besides going through `j`. On the other hand, in an implementation where `eq` and `eql` are not equivalent for `3`, then the particular `3` that is the value of `j` is an otherwise inaccessible part, but any other `3` is not. Thus `(setq foo 3)` is valid but `(setq foo j)` is erroneous. Since `(setq foo j)` is erroneous in some implementations, it is erroneous in all portable programs, but some other implementations may not be able to detect the error. (If this conclusion seems strange, it may help to replace `3` everywhere in the preceding argument with some obvious bignum such as `37537463883742489 8243` and to replace `10` with some even larger bignum.)

The `dynamic-extent` declaration should be used with great care. It makes possible great performance improvements in some situations, but if the user misdeclares something and consequently the implementation returns a pointer into the stack (or stores it in the heap), an undefined situation may result and the integrity of the Lisp storage mechanism may be compromised. Debugging these situations may be tricky. Users who have asked for this feature have indicated a willingness to deal with such problems; nevertheless, I do not encourage casual users to use this declaration.

Реализация может поддерживать другие спецификаторы деклараций (специфичные для неё). С другой стороны, компилятор Common Lisp может игнорировать некоторые виды деклараций (например, неподдерживаемые компилятором), за исключением спецификатора декларации `declaration`. Однако, разработчики компиляторов поощряются в том, чтобы выдавать предупреждение об использовании неизвестных деклараций.

9.3 Декларация типов для форм

Часто бывает полезно задекларировать, что значение, возвращаемое некоторой формой будет принадлежать определённому типу. Использование `declare` может применяться только к значениям связанным с переменной, но не для безымянных форм. Для этих целей определена специальная форма `the`. `(the` *`type form`*`)` означает, что значение формы *form* будет принадлежать типу *type*.

[Специальный оператор] **the** value-type form

Вычисляется форма *form*. То, что будет вычислено, будет возвращено из формы *the*. Дополнительно будет осуществлена проверка на принадлежность возвращённого значения типу *value-type* (его форма не выполняется) и в случае несовпадения выдана ошибка. (Реализация может и не осуществлять эту проверку. Однако такая проверка поощряется при работе в интерпретаторе.) В целом эта форма декларирует, что пользователь гарантирует, что значение формы всегда принадлежит заданному типу. Например:

```
(the string (copy-seq x))      ;Результат будет строкой
(the integer (+ x 3))          ;Результат + будет целым числом
(+ (the integer x) 3)          ;Значением x будет целое число
(the (complex rational) (* z 3))
(the (unsigned-byte 8) (logand x mask))
```

value-type may be any valid type specifier whatsoever. The point is that a type specifier need not be one suitable for discrimination but only for declaration.

In the case that the *form* produces exactly one value and *value-type* is not a `values` type specifier, one may describe a `the` form as being entirely equivalent to

(let ((#1=#:temp *form*)) (declare (type *value-type* #1#)) #1#)

A more elaborate expression could be written to describe the case where *value-type* is a `values` type specifier.

Глава 10

Символы

Lisp'овые символы является объектами данных, которые имеют три элемента, видимых для пользователя:

- *Список свойств* является списком, который позволяет хранить в символе именованные изменяемые данные.

- *Выводимое имя* должно быть строкой, которая является последовательностью строковых символов, идентифицирующей символ. Символы несут большую пользу, так как они могут быть обозначены просто заданным именем (например, напечатанным на клавиатуре). Выводимое имя изменять нельзя.

- *Ячейка пакета* должна ссылаться на объект пакета. Пакет является структурой данных, используемой для группирования имен символов. Символ уникально идентифицируется по имени, только когда рассматривается относительно пакета. Символ может встречаться в нескольких пакетах, но *родительским* пакетом может быть как максимум только один. Ячейка пакета ссылается на родительский пакет, если он есть. Ячейки пакетов обсуждаются в главе 11.

Символ может также содержать другие элементы, которые используются реализацией. Ещё одна важная функция, это использование символов в качестве имен переменных. Желательно, чтобы разработчик использовал такие элементы символа для реализации семантики переменных. Смотрите `symbol-value` и `symbol-function`.

Однако, существует несколько стратегий реализации, и такие возможные элементы символов здесь не описаны.

10.1 Список свойств

Начиная с самого создания, Lisp для каждого символа ассоциирует табличную структуру данных, называемую *список свойств* (для краткости *plist*). Список свойств содержит ноль и более элементов. Каждый элемент содержит ключ (называемым *индикатором*), который чаще всего является символом, и ассоциированным с ним значением (называемым иногда *свойством*), которое может быть любым Lisp'овым объектом. Среди индикаторов не может быть дубликатов. Список свойств может иметь только одно свойство для данного имени. Таким образом, значение может получено с помощью двух символов: исходного символа и индикатора.

Список свойств по целевому назначению очень похож на ассоциативный список. Различие в том, что список свойств является единственно подлинным объектом. Операции добавления и удаления элементов деструктивны, то есть при их использовании изменяется старый список, и нового списка свойств не создаётся. Ассоциативные список, наоборот, обычно изменяются недеструктивно (без побочных эффектов) с помощью добавления в начало новых элементов (смотрите `acons` и `pairlis`).

Список свойств реализуется, как ячейка памяти, содержащая список с чётным (возможно нулевым) количеством аргументов. (Обычно эта ячейка памяти является ячейкой списка свойств в символе, но в принципе подходит любая ячейка памяти, к которой можно применить `setf`) Каждая пара элементов в списке составляет строку. Первый в паре это индикатор, а второй — значение. Так как функции для списка свойств используют символ, а не сам список, то изменения этого списка свойств может быть записаны сохранением обратно в ячейку списка свойств символа. FIXME

Когда создаётся символ, его список свойств пуст. Свойства создаются с помощью `get` внутри формы `setf`.

Common Lisp не использует список свойств символа так интенсивно, как это делали ранние реализации Lisp'а. В Common Lisp'е нечасто используемые данные, такие как отладочная информация, информация

для компилятора и документация, хранятся в списках свойств.

[Функция] **get** *symbol indicator* **&optional** *default*

`get` ищет в списке свойств символа *symbol* индикатор равный `eq` индикатору *indicator*. Первый аргумент должен быть символом. Если такое свойство найдено, возвращается её значение. Иначе возвращается значение *default*.

Если *default* не указано, тогда для значения по-умолчанию используется `nil`.

Следует отметить, что способа отличить значения по-умолчанию и такое же значение свойства нет:

(get x y) ≡ (getf (symbol-plist x) y)

Допустим, что список свойств символа `foo` является `(bar t baz 3 hunoz "Huh?")`. Тогда, например:

(get 'foo 'baz) ⇒ 3
(get 'foo 'hunoz) ⇒ "Huh?"
(get 'foo 'zoo) ⇒ `nil`

`setf` может использоваться вместе с `get` для создания новой пары свойства, возможно замещая старую пару с тем же именем. Например:

(get 'clyde 'species) ⇒ `nil`
(setf (get 'clyde 'species) 'elephant) ⇒ elephant
и теперь (get 'clyde 'species) ⇒ elephant

В данном контексте может быть указан аргумент *default*. Он игнорируется в `setf`, но может быть полезен в таких макросах, как `push`, которые связаны с `setf`:

(push item (get sym 'token-stack '(initial-item)))

означает то же, что и

```
(setf (get sym 'token-stack '(initial-item))
      (cons item (get sym 'token-stack '(initial-item))))
```

а если упростить, то

```
(setf (get sym 'token-stack)
      (cons item (get sym 'token-stack '(initial-item))))
```

X3J13 voted in March 1989 to clarify the permissible side effects of certain operations; `(setf (get` *`symbol indicator`*`)` *`newvalue`*`)` is required to behave exactly the same as `(setf (getf (symbol-plist` *`symbol`*`)` *`indicator`*`)` *`newvalue`*`)`.

[Функция] **remprop** *symbol indicator*

Эта функция удаляет из символа *symbol* свойство с индикатором, равным `eq` индикатору *indicator*. Индикатор свойства и соответствующее значение удаляется из списка деструктивной склейкой списка свойств. Она возвращает `nil`, если указанного свойства не было, или не-`nil`, если свойство было.

(remprop x y) ≡ (remf (symbol-plist x) y)

Например, если список свойств `foo` равен

```
(color blue height 6.3 near-to bar)
```

тогда вызов

```
(remprop 'foo 'height)
```

вернёт значение не-`nil`, после изменения списка свойств символа `foo` на

```
(color blue near-to bar)
```

X3J13 voted in March 1989 to clarify the permissible side effects of certain operations; (`remprop` *`symbol indicator`*) is required to behave exactly the same as (`remf (symbol-plist` *`symbol`*`)` *`indicator`*).

[Функция] **symbol-plist** *symbol*

Эта функция возвращает список, который содержит пары свойств для символа *symbol*. Извлекается содержимое ячейки списка свойств и возвращается в качестве результата.

Следует отметить, что использование `get` с результатом `symbol-plist` *не* будет работать. Необходимо передавать в `get` символ, или же использовать `getf`.

С `symbol-plist` может использоваться `setf` для деструктивной замены списка свойств символа. Это относительно опасная операция, так как может уничтожить важную информацию, которую, возможно, хранила там реализация. Также, позаботьтесь о том, чтобы новый список свойств содержал чётное количество элементов.

[Функция] **getf** *place indicator* **&optional** *default*

`getf` ищет индикатор равный `eq` *indicator* в списке свойств, находящимся в *place*. Если он найден, тогда возвращается соответствующее значение. Иначе возвращается *default*. Если *default* не задан, то значение по-умолчанию используется `nil`. Следует отметить, что метода определения, вернулось ли значение по умолчанию или это значение свойства, нет. Часто *place* вычисляется из обобщённой переменной, принимаемой функцией `setf`.

`setf` может использоваться вместе `getf`, и в этом случае *place* должно быть обобщённым, чтобы его можно было передать в `setf`. Целью является добавление новой пары свойства, или изменению уже существующей пары, в списке свойств, хранящимся в *place*. В данном контексте может быть использован аргумент *default*. Он игнорируется функцией `setf`, но может быть полезен в таких макросах, как `push`, которые связаны с `setf`. Смотрите описание `get` для примера. X3J13 voted in March 1989 to clarify the permissible side effects of certain operations; `setf` used with `getf` is permitted to perform a `setf` on the *place* or on any part, *car* or *cdr*, of the top-level list structure held by that *place*.

X3J13 voted in March 1988 to clarify order of evaluation (see section 7.2).

[Макрос] **remf** place indicator

Эта функция в списке свойств, хранящимся в *place*, удаляет свойства, индикатор которого равен `eq` аргументу *indicator*. Индикатор свойства и соответствующее значение удаляется из списка деструктивной склейкой. `remf` возвращает `nil`, если свойства не было в списке, и некоторое не-`nil` значение, если свойство было. Форма *place* может быть любой обобщённой переменной, принимаемой `setf`. Смотрите `remprop`.

X3J13 voted in March 1989 to clarify the permissible side effects of certain operations; `remf` is permitted to perform a `setf` on the *place* or on any part, *car* or *cdr*, of the top-level list structure held by that *place*.

X3J13 voted in March 1988 to clarify order of evaluation (see section 7.2).

[Функция] **get-properties** *place indicator-list*

`get-properties` похожа на `getf` за исключением того, что второй аргумент является списком индикаторов. `get-properties` ищет первый встретившийся индикатор, который есть в списке индикаторов. Обычно *place* вычисляется из обобщённой переменной, которая может использоваться в `setf`.

`get-properties` возвращает три значения. Если было найдено свойство, то первые два значения являются индикатором и значением для первого свойства, чей индикатор присутствовал в списке *indicator-list*, и третье значение является остатком списка свойств, *car* элемент которого был индикатором (и соответственно *cadr* элемент — значением). Если ни одного свойства не было найдено, то все три значения равны `nil`. Третье значение является флагом успешности или неудачи, и позволяет продолжить поиск свойств в оставшемся списке.

10.2 Выводимое имя

Каждый символ имеет ассоциированную строку, называемую *выводимое имя*. Строка используется для вывода отображения символа: если символы в строке будут напечатаны в функцию `read` (с необходимым, где надо, экранированием), они будут интерпретированы как ссылка на

этот символ (если он интернирован). Если символ выводится куда-либо, `print` печатает его выводимое имя. Для подробностей, смотрите раздел о *читателе* (раздел 22.1.1) и *писателе* (раздел 22.1.6).

[Функция] **symbol-name** *sym*

Данная функция возвращает выводимое имя для символа *sym*. Например:

(symbol-name 'xyz) ⇒ "XYZ"

Это не очень хорошая идея, изменять строку, которая используется, как выводимое имя символа. Такое изменение может сильно запутать функцию `read` и систему пакетов.

Изменение строки, которая служит именем символа, является ошибкой.

10.3 Создание символов

Символы бывают двух видов. Интернированный символ – это символ, который может быть задан с помощью имени и каталога (*пакета*) и непосредственно имени символа.

A request to locate a symbol with that print name results in the same (`eq`) symbol. Every time input is read with the function `read`, and that print name appears, it is read as the same symbol. This property of symbols makes them appropriate to use as names for things and as hooks on which to hang permanent data objects (using the property list, for example).

Обращение к символу с некоторым именем возвращает символ, равный `eq` необходимому. Каждый раз когда функция `read` встречает строку для символа, они находит соотвествующий данной строке символ. FIXME

Интернированные символы обычно создаются автоматически. Во время первого обращения к символу в системе пакетов (с помощью `read`, например), данный символ создаётся автоматически. Для обращения к интернированному символу используется функция `inter`, или другая с ней связанная.

Несмотря на то, что интернированные символы используются чаще всего, они не будут больше здесь рассматриваться. Для подробной информации смотрите главу 11.

Дезинтернированный символ является символом, используемым в качестве объекта данных, без связи с каталогом (он не имеет родительского пакета). Неинтернированный символ выводится как `#:` с последующим выводимым именем.

[Функция] **make-symbol** *print-name*

`(make-symbol` ***print-name***`)` создаёт новый неинтернированный символ, у которого выводимое имя является строкой *print-name*. Полученный символ не имеет значения функции (unbound) и пустой список свойств.

Строка, переданная в аргументе, может использоваться сразу, либо сначала копироваться. Это зависит от реализации. Пользователь не может полагаться на то, что `(symbol-name (make-symbol x))` равно `eq x`, но и также не может изменять строку переданную в качестве аргумента для `make-symbol`.

Заметка для реализации: Реализация может, например, скопировать строку в область «только для чтения», полагаясь на то, что строка никогда не будет изменена.

[Функция] **copy-symbol** *sym* **&optional** *copy-props*

Эта функция возвращает новый неинтернированный символ с тем же выводимым именем, что *sym*.

X3J13 voted in March 1989 that the print name of the new symbol is required to be the same only in the sense of `string=`; in other words, an implementation is permitted (but not required) to make a copy of the print name. User programs should not assume that the print names of the old and new symbols will be `eq`, although they may happen to be `eq` in some implementations.

Если *copy-props* не-`nil`, тогда начальное значение и определение функции нового символа будут те же, что и в переданном *sym*, а список свойств будет скопирован из исходного символа.

При использовании данной функции копируются cons-ячейки только верхнего уровня. It is as if `(copy-list (symbol-plist` ***sym***`))` were used as the property list of the new symbol.

Если *copy-props* является `nil` (по-умолчанию), тогда новый символ не будет связан, не будет иметь определения функции, и список свойств будет пуст.

[Функция] **gensym &optional** *x*

`gensym` создаёт выводимое имя и создаёт новый символ с этим именем. Она возвращает новый неинтернированный символ.

Созданное имя содержит префикс (по-умолчанию `G`), с последующим десятичным представлением числа.

`gensym` обычно используется для создания символа, который не виден пользователю, и его имя не имеет важности. Необязательный аргумент используется нечасто. Имя образовано от «генерация символа», и символы созданные, таким образом, часто называются «gensyms».

Если необходимо, чтобы сгенерированные символы были интернированными и отличными от существующих символов, тогда удобно использовать функцию `gentemp`.

X3J13 voted in March 1989 to alter the specification of `gensym` so that supplying an optional argument (whether a string or a number) does *not* alter the internal state maintained by `gensym`. Instead, the internal counter is made explicitly available as a variable named `*gensym-counter*`.

If a string argument is given to `gensym`, that string is used as the prefix; otherwise "`G`" is used. If a number is provided, its decimal representation is used, but the internal counter is unaffected. X3J13 deprecates the use of a number as an argument.

[Переменная] ***gensym-counter***

Переменная хранит состояние счётчика для функции `gensym`. `gensym` использует десятичное представление этого значения в качестве части имени генерируемого символа, а затем наращивает этот счётчик.

Первоначальное значение этой переменной зависит от реализации, но должно быть неотрицательным целым.

Пользователь в любое время может присваивать или связывать это переменную, но значение должно быть неотрицательным целым.

[Функция] **gentemp &optional** *prefix package*

`gentemp`, как и `gensym`, создаёт и возвращает новый символ. `gentemp` отличается от `gensym` в том, что возвращает интернированный символ

(смотрите `intern`) в пакете *package* (который по-умолчанию является текущим, смотрите `*package*`). `gentemp` гарантирует, что символ будет новым, и не существовал ранее в указанном пакете. Она также использует счётчик, однако если полученный символ уже существует счётчик наращивается, и действия повторяются, пока не будет найдено имя ещё не существующего символа. Сбросить счётчик невозможно. Кроме того, префикс для `gentemp` не сохраняется между вызовами. Если аргумент *prefix* опущен, то используется значение по-умолчанию `T`.

[Функция] **symbol-package** *sym*

Для заданного символа возвращает содержимое ячейки пакета. Результат может быть объектом пакета или `nil`.

[Функция] **keywordp** *object*

Аргумент может быть любым Lisp'овым объектом. Предикат `keywordp` истинен, если аргумент является символом и этот символ является ключевым (значит, что принадлежит пакету ключевых символов). Ключевые символы — это символы, которые записываются с двоеточием в начала. Каждый ключевой символ является константой, в смысле, что выполняются сами в себя. Смотрите `constantp`.

Глава 11

Пакеты

В ранних реализациях Lisp'а использовалось одно пространство имен для всех символов. В больших Lisp'овых системах, с модулями, написанными разными программистами, случайные совпадения имен стали серьёзной проблемой. Common Lisp решает эту проблему с помощью *системы пакетов*, производной от системы пакетов, разработанной для Lisp Machine [55]. Кроме того система пакетов делает модульную структуру больших Lisp систем более явной.

Пакет — это структура данных, которая устанавливает связь между выводимыми именами (строкой) и символами. Во время работы только один пакет является текущим, и этот пакет используется Lisp'овым считывателем при преобразовании строк в символы. Текущий пакет храниться в глобальной переменной `*package*`. С помощью *имени пакета* в выводимом имени символа существует возможность ссылаться на символы других пакетов. Например, когда `foo:bar` будет прочтён Common Lisp'ом, то будет ссылаться на символ `bar` из пакета `foo`. (Но при условии, что `bar` является экспортированным символом из `foo`, то есть символом, который является видимым извне пакета `foo`. Ссылка на внутренний символ требует удваивания двоеточия: `foo::bar`.)

Связь строк и символов, доступных в данном пакете, делится на два вида: *внешняя* и *внутренняя*. Речь идёт о символах доступных с помощью этих связей, как о *внешних* и *внутренних* символах пакета, хотя на самом деле различаются связи, а не символы. Внутри заданного пакета имя ссылается максимум на один символ. Если оно ссылается на символ, тогда этот символ в данном пакете является или внешним, или внутренним.

Внешние символы являются частью интерфейса пакета, доступного для других пакетов. Внешние символы должны быть аккуратно спроектированы и предоставлены для пользователей этого пакета. Внутренние символы предусмотрены только для внутреннего использования, и эти символы обычно скрыты от других пакетов. Большинство символов создаются как внутренние. Они становятся внешними, только если явно передаются в команду `export`.

Символ может встречаться во многих пакетах. Он будет всегда иметь одно и то же имя, но в некоторых пакетах может быть внешним, а в других внутренним. С другой стороны, одно и то же имя (строка) может ссылаться на различные символы в различных пакетах.

Обычно, символ, который встречается в одном и более пакетах, будет иметь только один родительский пакет, называемый *домашний пакет* символа. Говорится, что такой пакет владеет символом. Все символы содержат компонент, называемый *ячейка пакета*, который хранит указатель на домашний пакет. Символ, который имеет домашний пакет, называется *интернированным*. Некоторые символы не имеют домашнего пакета. Они называются *дезинтернированными*. Их ячейка пакета содержит значение `nil`.

Пакеты могут быть представлены как слои. С этой точки зрения, для пользователя пакет выглядит как коллекция связей строк с внутренними и внешними символами. Некоторые из этих связей могут устанавливаться внутри самого пакета. Другие связи наследуются из других пакетов с помощью конструкции `use-package`. (Механизм такого наследования описан ниже.) В дальнейшем, мы будем называть символ *доступным* в пакете, если на него можно сослаться без указания имени пакета, вне зависимости от того унаследована ли связь символов с именем или установлена текущим пакетом. И мы будем называть символ *родным* в пакете, если связь установлена самим пакетом, а не унаследована. Таким образом, *родной* символ в пакете является *доступным*, но *доступный* символ не обязательно является *родным*.

Символ называется *интернированным в пакет*, если он доступен в этом пакете и имеет некоторый родительский пакет. Обычно все символы доступные в пакете будут в собственности некоторого пакета, но иногда описываются случаи доступности никому не принадлежащего (*дезинтернированного*) символа.

«*Интернировать* символ в пакет!» означает сделать так, чтобы пакет стал владельцем символа, если до этого было не так. Этот процесс

выполняется функцией `intern`. Если символ прежде был бесхозным (никому не принадлежал), тогда пакет становится его владельцем (домашним пакетом). Но если символ уже принадлежал кому-то, то домашний пакет не меняется.

«*Дезинтернировать* символ из пакета» означает убрать его из пакета. Данный процесс выполняется функцией `unintern`. FIXME

11.1 Правила согласованности

Ошибки связанные с пакетами могут быть очень тонкими и запутанными. Система пакетов Common Lisp'а спроектирована с рядом безопасных мер для предотвращения большинства распространённых ошибок, которые могли бы быть при обычном использовании. Может показаться, что защита используется излишне, однако опыт предыдущих систем пакетов показал, что такие меры необходимы.

При работе с системой пакетов, полезно держать в памяти следующие правила. Эти правила остаются в силе, пока пользователь не изменил значение `*package*`.

- *Согласованность чтения-чтения:* Чтение одного и того же имени приводит к одному и тому же символу (`eq` вернёт истину).
- *Согласованность вывода-чтения:* Интернированный символ всегда выводится как последовательность строковых символов, которые при повторном чтении дают исходный символ (`eq` вернёт истину).
- *Согласованность вывода-вывода:* Если два интернированных символа не равны `eq`, тогда их печатаемые отображения будут различными последовательностями строковых символов.

Эти правила согласованности остаются в силе, несмотря на любое количество неявных случаев интернирования в Lisp'овых формах, загрузках файлов и так далее. Пока текущий пакет не меняется, согласованность сохраняется вне зависимости от порядка загрузки файлов или истории вводимых символов. Правила могут быть нарушены только явным действием: изменением значения `*package*`, продолжением выполнения после ошибки, или вызовом одной из

«опасных» функций `unintern`, `unexport`, `shadow`, `shadowing-import` или `unuse-package`.

11.2 Имена пакетов

Каждый пакет имеет имя (строку) и, возможно, несколько псевдонимов. Они указываются при создании пакета, и могут быть изменены позднее. Имя пакета должно быть длинным и информативным, например `editor`. Псевдоним должен быть коротким и простым в написании, например `ed`.

Для имен пакетов существует только одно пространство имён. Функция `find-package` транслирует имя или псевдоним пакета в объект пакета. Функция `package-name` возвращает имя пакета. Функция `package-nicknames` возвращает список всех псевдонимов для пакета. Функция `rename-package` заменяет текущее имя пакета и псевдонимы на указанные пользователем. Переименование пакета изредка бывает полезным. Но один из случаев, например, для разработки, когда необходимо загрузить две версии одного пакета в Lisp систему. Можно загрузить первую версию, переименовать её, и затем загрузить другую версию, без разрешения конфликтов имен.

Когда Lisp считыватель встречает полное имя символа, он обрабатывает часть имени пакета, также как и часть имени символа, возводя все неэкранированные строковые символы в верхний регистр. Экранирование строковых символов производится с помощью символов `\` или `|`. Поиск, осуществляемый функцией `find-package`, является регистрозависимым, также как и для символов. Следует отметить, что `|Foo|:|Bar|` ссылается на символ, имя которого `Bar`, в пакете `Foo`. Для сравнения `|Foo:Bar|` ссылается на семизначный символ, имя которого содержит двоеточие (а также две заглавные и четыре прописные буквы) и интернирован в текущий пакет. В данной книге символы и пакеты указываются без экранирования строковыми символами только в нижнем регистре, при этом внутри Lisp машины они будут переведены в верхний регистр.

Большинство функций, которые принимают имя пакета, могут принимать или символ, или строку. Если указан символ, то используется его выводимое имя, которое подвергается обычным преобразованиям в верхний регистр. Если указана строка, то она должна быть преобразована для полного совпадения с именем пакета.

X3J13 voted in January 1989 to clarify that one may use either a package object or a package name (symbol or string) in any of the following situations:

- the `:use` argument to `make-package`
- the first argument to `package-use-list`, `package-used-by-list`, `package-name`, `package-nicknames`, `in-package`, `find-package`, `rename-package`, or `delete-package`,
- the second argument to `intern`, `find-symbol`, `unintern`, `export`, `unexport`, `import`, `shadowing-import`, or `shadow`
- the first argument, or a member of the list that is the first argument, to `use-package` or `unuse-package`
- the value of the *package* given to `do-symbols`, `do-external-symbols`, or `do-all-symbols`
- a member of the *package-list* given to `with-package-iterator`

Note that the first argument to `make-package` must still be a package name and not an actual package; it makes no sense to create an already existing package. Similarly, package nicknames must always be expressed as package names and not as package objects. If `find-package` is given a package object instead of a name, it simply returns that package.

11.3 Преобразование строк в символы

Значение специальной переменной `*package*` должно быть всегда объектом пакета (не именем). Данный объект называется *текущим пакетом*.

Когда Lisp'овый считыватель получает строку для символа, он ищет его имя в текущем пакете. Данный поиск может привести к поиску в других пакетах, экспортированные символы которых унаследованы текущим пакетом. Если имя найдено, то возвращается соответствующий символ. Если имя не найдено (то есть, в текущем пакете не существует соответствующего доступного символа), то создаётся новый символ и помещается в текущий пакет. Если точнее, то текущий пакет становится владельцем (домашним пакетом) символа. Если это имя будет прочитано

ещё раз позже и в этом же пакете, то будет возвращён уже созданный символ.

Часто необходимо сослаться на внешний символ в некотором другом, не текущем пакете. Это может быть сделано с помощью *полного имени*, включающем имя пакета, затем двоеточие, и, наконец, имя символа. Это приводит к поиску символа в указанном, а не текущем пакете. Например, `editor:buffer` ссылается на внешний символ с именем `buffer` доступный из пакета с именем `editor`, вне зависимости от того, есть ли в текущем пакете символ с таким же именем. Если пакета с именем `editor` или символа с именем `buffer` в указанном пакете не сущетсвует, Lisp'овый считыватель сигнализирует исправимую ошибку.

В редких случаях пользователь может нуждаться в ссылке на *внутренний* символ некоторого не текущего пакета. Это нельзя сделать с помощью двоеточия, так как данная запись позволяет ссылаться только на внешние символы. Однако, это можно сделать с помощью двойного двоеточия `::`, указанного вместо одинарного. Если используется `editor::buffer`, то эффект такой же, как если бы произошла попытка найти символ с именем `buffer` и `*package*` была связана с объектом с именем `editor`. Двойное двоеточие должно использоваться с осторожностью.

Пакет с именем `keyword` содержит все ключевые символы используемые Lisp'овой системой и пользовательским кодом. Такие символы должны быть легко доступны из любого пакета, и конфликт имён не является проблемой, так как эти символы используются только в качестве меток и не содержат значений. Так как ключевые символы используются часто, то Common Lisp для них предоставляет специальный синтаксис. Любой символ с двоеточием в начале и без имени пакета (например `:foo`) добавляется (или ищется) в пакете `keyword` как *внешний* символ. Пакет `keyword` также отличается тем, что при добавлении в него символа, последний автоматически становится внешним. Символ также автоматически декларируется как константа (смотрите `defconstant`) и его значением становится он сам. В целях стиля, ключевые символы должны всегда быть доступны с помощью двоеточия в начале имени. Пользователь никогда не должен импортировать или наследовать ключевые символы в другие пакета. Попытка использовать `use-package` для `keyword` пакета является ошибкой.

Каждый символ содержит ячейку пакета, которая используется

для записи домашнего пакета символа, или `nil`, если пакет неинтернированный. Эта ячейка доступна с помощью функции `symbol-package`. Когда интернированный символ печатается, если это символ в пакете ключевых символов, тогда он выводится с двоеточием в начале, иначе, если он доступен (напрямую или унаследовано) в текущем пакете, он печатается без имени пакета, иначе он печатается полностью, с именем пакета, именем символа и `:` в качестве разделителя для внешнего символа, и `::` для внутреннего.

Символ, у которого слот (ячейка) пакета содержит `nil` (то есть, домашний пакет отсутствует) печатается с `#:` в начале имени. С использованием `import` и `unintern` возможно создать символ, который не имеет домашнего пакета, но фактически доступен в некоторых пакетах. Lisp система не проверяет такие патологические случаи, и такие символы будут всегда печататься с предшествующими `#:`.

В целом, синтаксис имен символов может быть выражен в следующих четырёх примерах.

`foo:bar` При прочтении, выполняется поиск символа `BAR` среди внешних символов пакета `FOO`. Такой вывод, когда символ `bar` является внешним в домашнем пакете `foo` и недоступен в текущем пакете.

`foo::bar` При прочтении, интернирует символ `BAR`, как если бы пакет `FOO` являлся текущим. Такой вывод, когда символ `bar` является внутренним в его домашнем пакете `foo` и недоступен в текущем пакете.

`:bar` При прочтении, интернирует `BAR` как внешний символ в пакете `keyword` и выполняет его самого в себя. Такой вывод, когда `keyword` является домашним пакетом для символа.

`#:bar` При прочтении, создаёт новый дезинтернированный символ с именем `BAR`. Такой вывод, когда символ `bar` не имеет домашнего пакета.

Все другие использования двоеточия внутри имен символов не определены Common Lisp'ом, но зарезервированы для реализаций. Сюда включены имена с двоеточием в конце, или содержащими два и более двоеточиях или просто состоящими из двоеточия.

11.4 Экспортирование и импортирование символов

Символы из одного пакета могут стать доступными в другом пакете двумя способами.

Первый способ, каждый отдельный символ может быть добавлен в пакет с использованием функции `import`. Форма `(import 'editor:buffer)` принимает внешний символ с именем `buffer` в пакете `editor` (этот символ распознаётся Lisp считывателем) и добавляет его в текущий пакет в качестве внутреннего символа. Символ становится *доступным* в текущем пакете. Импортированный символ автоматически не экспортируется из текущего пакета, но если он уже существовал в пакете и был внешним, то это свойство не меняется. После вызова `import` в импортирующем пакете появляется возможность ссылаться на `buffer` без указания полного имени. Свойства `buffer` в пакете `editor` не меняются, `editor` продолжает оставаться домашним пакетом для этого символа. Будучи импортированным, символ будет присутствовать в пакете и может быть удалён только с помощью вызова `unintern`.

Если символ уже присутствует в импортирующем пакете, `import` ничего не делает. Если другой символ с таким же именем `buffer` уже доступен в импортирующем пакете (напрямую или унаследован), тогда сигнализируется исправимая ошибка, как написано в разделе 11.5. `import` не допускает сокрытия одного символа другим.

Символ называется скрытым другим символом в некотором пакете, когда первый символ был бы доступен через наследование, если бы не присутствие второго символа. Для импортирования символа без ошибки скрытия, используйте функцию `shadowing-import`. Она вставляет символ в указанный пакет, как внутренний символ, вне зависимости от того, происходит ли скрытие другого символа с тем же именем. Если другой символ с тем же именем присутствовал в пакете, тогда этот символ сначала удаляется из пакета `unintern`. Новый символ добавляется в список затеняющих символов FIXME. `shadowing-import` должна использоваться аккуратно. Она изменяем состояние системы пакетов так, что правила согласованности могут перестать работать.

Второй способ предоставляется функцией `use-package`. Эта функция делает так, что пакет наследует все внешние символы некоторого другого пакета. Эти символы становятся доступными, как *внутренние*

символы, используемого пакета. То есть, на них можно ссылаться без указания пакета внутри текущего пакета, но они не становятся доступными в других пакетах, использующих данный. Следует отметить, что `use-package`, в отличие от `import`, не делает новые символы *родственными* в текущем пакете, а делает их только *доступными* с помощью наследования связи символов с именами. `use-package` проверяет конфликты имен между импортируемыми и уже доступными символами в импортирующем пакете. Это подробнее описано в разделе 11.5.

Обычно пользователь, по-умолчанию работая в пакете `common-lisp-user`, будет загружать ряд пакетов в Lisp систему для предоставления расширенного рабочего окружения, и затем вызывать `use-package` для каждого из этих пакетов для простого доступа к их внешним символами. `unuse-package` производит обратные действия относительно `use-package`. Внешние символы используемые пакетом перестают наследоваться. Однако, любые импортированные символы, остаются доступными.

Не существует способа наследовать *внутренние* символы другого пакета. Для ссылки на внутренний символ, пользователь должен поменять домашний пакет для данного символа на текущий, или использовать полное имя (вместе с пакетом), или импортировать этот символ в текущий пакет.

Различие между внешними и внутренними символами прежде всего означает скрытие имен, так чтобы одна программа не могла использовать пространство имен другой программы.

Когда `intern` или некоторые другие функций хотят найти символ в заданном пакете, они сначала ищут символ среди внешних и внутренних символов текущего пакета, затем они в неопределённом порядке ищут среди внешних символов используемых пакетов. Порядок не имеет значение. В соответствии с правилами разрешения конфликтов имен (смотрите ниже), если конфликтующие символы существуют в двух и более пакетах, унаследованных пакетом X, символ с этим именем должен также быть в X, как затеняющий символ. Конечно, реализации могут выбрать другой, более эффективный способ для реализации такого поиска, не изменяя поведение ранее описанного интерфейса.

Функция `export` принимает символ, который доступен в некотором указанном пакете (напрямую или унаследован из другого пакета) и делает его внешним символом этого пакета. Если символ уже доступен

как внешний, `export` ничего не делает. Если символ представлен, как внутренний *родственный* символ, его статус просто меняется на внешний. Если он доступен, как внутренний символ, полученный с помощью `use-package`, символ сначала импортируется в пакет, а затем делается внешним. (После этого символ становиться импортированным в пакет и остаётся в нем, вне зависимости от того будет ли использована `unuse-package` или нет). Если символ вообще недоступен в указанном пакете, то сигнализируется ошибка с возможностью решить проблему, а, именно, какой символ должен быть импортирован.

Функция `unexport` откатывает ошибочное экспортирование символа. Она работает только для символов напрямую представленных в текущем пакете, меняя их свойство на «внутреннее». Если `unexport` получает символ, который в текущем пакете уже является внутренним, она ничего не делает. Если получает вообще недоступный символ, то сигнализирует ошибку.

11.5 Конфликты имён

Основное неизменяемое правило системы пакетов состоит в том, что внутри одного пакета каждое имя может ссылаться не более, чем на один символ. *Конфликтом имён* называется ситуация, когда существует более одного подходящего символа и не ясно какой из них должен быть выбран. Если система не будет следовать одному методу выбора, то правило чтения-чтения будет нарушено. Например, некоторая программа или данные должны быть прочитаны при некоторой связи имени с символом. Если связь изменяется в другой символ, и затем прочитывается дополнительная программа или символ, то две программы не будут получать доступ к одному и тому же символу, даже если используют одинаковое имя. Даже если система всегда выбирает один метод выбора символа, конфликт имён приводит к связи имени с символом отличной от той, что ожидает пользователь, приводя к некорректному выполнению программы. Таким образом, в любом случае возникновения конфликта имён сигнализируется ошибка. Пользователь может указать системе пакетов, как разрешить конфликт, и продолжить выполнение.

Может быть ситуация, когда один символ может быть доступен для пакета более чем одним способом. Например, символ может быть

внешним символов более чем одного используемого пакета, или символ может быть напрямую представлен в пакете и также унаследован из другого пакета. В таких случаях конфликта имён не возникает. Один и тот же символ не может конфликтовать сам с собой. Конфликты имён возникают только между разными символами с одинаковыми именами.

Создатель пакета может заранее указать системе, как разрешать конфликт имён, используя *затенения*. Каждый пакет имеет список затеняющих символов. Затеняющий символ имеет преимущество перед любым другим символом с тем же именем. Разрешение конфликта с участием затеняющего символа всегда происходит в пользу последнего без сигнализирования об ошибке (за исключение одного использования `import` описанного ниже). Функции `shadow` и `shadowing-import` могут использоваться для декларации затеняющих символов.

Конфликты имён обнаруживаются, когда они становятся возможными, то есть, когда изменяется структура пакета. Нет смысла проверять конфликты в каждом процессе поиска имени для символа.

Функции `use-package`, `import` и `export` проверяют возникновение конфликтов имён. `use-package` делает внешние символы пакета доступными для использования в другом пакете. Каждый из этих символов проверяется на конфликт имён с уже доступным в пакете символом. `import` добавляет один символ как внутренний символ пакета, проверяя на конфликты имён с *родственными* или *доступными символами*. `import` сигнализирует об ошибке конфликта имён, даже если конфликт произошёл с затеняющим символом. Это объясняется том, что пользователь дал два явных и взаимоисключающих указания. `export` делает один символ доступным для всех пакетов, которые используют пакет, из которого символ был экспортирован. Все эти пакеты проверяются на конфликты имён: `(export` *`s p`*`)` делает `(find-symbol (symbol-name` *`s q`*`)` для каждого пакета *q* в `(package-used-by-list` *`p`*`)`. Следует отметить, что в обычно при выполнении `export` в течение первоначального определения пакета, результат `package-used-by-list` будет `nil` и проверка конфликта имён не будет занимать много времени.

Функция `intern`, которая является одной из часто используемых Lisp'овыми считывателем для поиска имен символов, не нуждается в проверке конфликта имён, потому что она никогда не создаёт новые символы, если символ с указанным именем уже *доступен*.

`shadow` и `shadowing-import` никогда не сигнализируют ошибку

конфликта имён, потому что пользователь, вызывая их, указывает, как возможный конфликт будет разрешён. `shadow` проверяет конфликт имён при условии, что другой существующий символ с указанным именем доступен и, если так, является ли *родственным* или унаследованным. В последнем случае, новый символ создаётся, чтобы затенить старый. `shadowing-import` проверяет конфликт имён при условии, что другой существующий символ с указанным именем доступен и, если так, он затеняется новым символом, что означает, что он должен быть дезинтернирован, если он был представлен в пакете напрямую.

`unuse-package`, `unexport` и `unintern` (когда символ, будучи дезинтернированым, не является затеняющим символом) не требуют проверки конфликтов имён, потому что они просто удаляют символы из пакета. Они не делают какие-либо новые символы доступными.

Указание затеняющего символа в функцию `unintern` может раскрывать конфликт имён, который был ранее разрешён с помощью затенения. Если пакет A использует пакеты B и C, A содержит затеняющий символ `x`, и B, и C, каждый содержит внешний символ с именем `x`, тогда при удалении затеняющего символа `x` из A будет обнаружен конфликт между `b:x` и `c:x`, если эти два символа различны. В этом случае `unintern` будет сигнализировать ошибку.

Прерывание ошибки конфликта имён оставляет оригинальный символ доступным. Функции для пакетов всегда сигнализируют ошибки конфликтов имён перед любыми изменениями в структуре пакетов. Однако когда изменяются много символов за раз, например, когда `export` получила список символов, реализация может обрабатывать каждое изменение по отдельности, таким образом прерывание ошибки, возникшей для второго символа, не приведёт к отмене результатов, выполненных для первого символа. Однако, прерывание из ошибки конфликта имён при использовании `export` для одного символа не оставит этот символ доступным для одних пакетов и недоступным для других. В отношении к каждому обрабатываемому символу `export` ведёт себя как атомарная операция.

Продолжение из ошибки конфликта имён должно предлагать пользователю возможность разрешить конфликт имён в пользу какого-либо из кандидатов. Структура пакета должна измениться в соответствие с разрешением конфликта имён, с помощью `shadowing-import`, `unintern` или `unexport`.

Конфликт имён в `use-package` между *родственными* символом

в использующем пакете и внешними символом в используемом пакете может быть разрешён в пользу первого символа с помощью создания затеняющего символа, или в пользу второго символа с помощью дезинтернирования первого символа из использующего пакета. Последний способ опасен, если символ для дезинтернирования является внешним символом использующего пакета, так как перестанет быть внешним символом.

Конфликт имён в `use-package` между двумя внешними символами, унаследованными в использующем пакете из других пакетов, может быть разрешён в пользу одного из символов, с помощью импортирования его в использующий пакет и превращения его в затеняющий символ.

Конфликт имён в `export` между символом для экспорта и символом уже присутствующем в пакете, который будет наследовать свеже экспортирующийся символ, может быть разрешён в пользу экспортируемого символа с помощью дезинтернирования другого символа, или в пользу уже присутствующего символа, превращением его в затеняющий.

Конфликт имён в `export` или `unintern` из-за пакета, наследующего два различных символа с одинаковым именем из двух разных пакетов, может быть разрешён в пользу одного из символов с помощью импортирования его в использующий пакет и превращении его в затеняющий, также как и при использовании `use-package`.

Конфликт имён в `import` между символом для импортирования и символом унаследованным из некоторого другого пакета может быть разрешён в пользу импортируемого символа превращением его в затеняющий символ или в пользу уже доступного символа с помощью отмены `import`. Конфликт имён в `import` с символом уже присутствующем в пакете может быть разрешён с помощью дезинтернирования этого символа, или отмены `import`.

Хороший стиль пользовательского интерфейса диктует то, что `use-package` и `export`, которые могут вызывать за раз много конфликтов имён, сначала проверяют все конфликты имён перед тем, как предоставить любой из них пользователю. Пользователь может выбрать разрешать ли конфликт для всех сразу или по отдельности. Последний способ трудоёмок, но позволяет разрешить каждый конфликт отдельным способом.

Реализации могут предлагать другие пути решения конфликтов имён. Например, если конфликтующие символы, не используемые для

объектов, а только для имен функций, они могут быть «слиты» с помощью размещения определения функции в обоих символах. Ссылка на любой символ в целях вызова функции будет эквивалентна. Похожая операция «слияния» может быть сделана для значений переменных или для вещей, сохранённых в списке свойств. В Lisp Machine Lisp'е, например, можно было также *выдвинуть (forward)* значение, функцию и ячейки свойств так, что изменения в одном символе приводили к изменению в другом. Некоторые другие реализации позволяют сделать это для ячейки значения, но на для ячейки списка свойств. Тогда пользователь может знать является ли этот метод разрешения конфликтов имён адекватным, потому что метод будет работать только, если использования двух не-`eq` символов с одинаковым именем не будет препятствовать корректной работе программы. Значение стоимости слияния символов, как разрешении конфликта имён, в том, что оно может избегать необходимости выбрасывать весь Lisp'овый мир, исправлять формы определения пакета, который вызвал ошибку и начинать с нуля.

11.6 Системные пакеты

`common-lisp` Пакет с именем `common-lisp` содержит базовые элементы системы ANSI Common Lisp. Его внешние символы включают все видимые пользователю функции и глобальные переменные, которые представлены в ANSI Common Lisp системе, такие как `car`, `cdr` и `*package*`. Следует однако отметить, что домашним пакетом этих символов не обязательно должен является пакет `common-lisp` (для символов, например `t` и `lambda` будет проще быть доступными и в `common-lisp`, и в другом пакете, возможно с именем `lisp` FIXME). Почти все другие пакеты должны использовать `common-lisp`, таким образом эти символы будут доступны без использования полного имени (с двоеточием). Данный пакет имеет псевдоним `cl`.

`common-lisp-user` Пакет `common-lisp-user` является, по-умолчанию, текущим пакетом во время запуска ANSI Common Lisp системы. Этот пакет использует пакет `common-lisp` и имеет псевдоним `cl-user`. В зависимости от реализации он также может содержать

другие символы и использовать платформоспецифичные пакеты.

`keyword` Этот пакет содержит все ключевые символы, используемые встроенными или пользовательскими Lisp'овыми функциями. Имя символа, начинающегося с двоеточия, интерпретируется, как ссылка на символ из этого пакета, который всегда является внешним. Все символы в этом пакете является константами, которые вычисляются сами в себя, поэтому пользователь может записывать `:foo` вместо `':foo`.

Пользовательским программам не могут выполнять следующие действия, которые могут противоречить базовому функционалу или негативно с ним взаимодействовать. Если над символом в пакете `common-lisp` совершается одно из действий ниже, за исключением явных указаний в книге, последствия не определены.

- связывание или изменение значения символа (лексически или динамически)
- определение или связывания символа с функцией
- определение или связывания символа с макросом
- определение его как спецификатора типа (`defstruct`, `defclass`, `deftype`)
- определение его как структуры (`defstruct`)
- определение его как декларации
- определение его макроса символа
- изменение выводимого имени символа
- изменение пакета символа
- трассировка символа
- декларация или прокламация символа как специального или лексического
- декларация или прокламация типа символа (`type`, `ftype`)

- удаление символа из пакета `common-lisp`
- определение как макроса компилятора

Если такой символ не определён глобально как переменная или константа, пользовательская программа может связывать его лексически и декларировать тип `type` данного связывания.

Если такой символ не определён как функция, макрос или специальная форма, пользовательская программа может лексически связать символ с функцией, декларировать тип `ftype` данного связывания и трассировать это связывание.

Если такой символ не определён как функция, макрос или специальная форма, пользовательская программа может лексически связать символ с макросом.

В качестве примера, поведение данного фрагмента кода

```
(flet ((open (filename &key direction)
         (format t "~%OPEN was called.")
         (open filename :direction direction)))
  (with-open-file (x "frob" :direction ':output)
    (format t "~%Was OPEN called?")))
```

не определено. Даже в «разумной» реализации, например, раскрытие макрос `with-open-file` должно ссылаться на функцию `open`. Тем не менее, предшествующие правила определяют является ли реализация «разумной». Данный фрагмент кода нарушает правила. Официально его поведение, таким образом, полностью не определено, и точка.

Следует отметить, что «изменение списка свойств» не входит в список упомянутых действий, таким образом пользовательская программа может добавлять или удалять свойства для символов в `common-lisp` пакете.

11.7 Функции и переменные для системы пакетов

Заметка для реализации: In the past, some Lisp compilers have read the entire file into Lisp before processing any of the forms. Other compilers have arranged for

the loader to do all of its intern operations before evaluating any of the top-level forms. Neither of these techniques will work in a straightforward way in Common Lisp because of the presence of multiple packages.

Для описанных здесь функций, все необязательные аргументы с именем *package* имеют значение по-умолчанию `*package*`. Там, где функция принимает аргумент, который может быть символом или списком символов, значение `nil` расценивается, как пустой список символов. Любой аргумент, описанный как имя пакета, может быть символом или строкой. Если указан символ, то используется его выводимое имя. Если строка, то пользователь должен позаботиться о преобразовании регистра символов в верхний там, где это необходимо.

[Переменная] ***package***

Значение этой переменной должно быть объектом пакета. Этот пакет называется текущим. Первоначальное значение `*package*` является пакетом `common-lisp-user`.

Функции `load` и `compile-file` пересвязывают `*package*` в текущее значение. Если некоторая форма в файле во время загрузки изменяет значение `*package*`, старое значение будет восстановлено после завершения загрузки.

Функция `compile-file` пересвязывает `*package*` в текущее значение. Если некоторая форма в файле во время загрузки или компиляции изменяет значение `*package*`, старое значение будет восстановлено после завершения функции.

[Функция] **make-package** *package-name* **&key** *:nicknames :use*

Эта функция создаёт и возвращает новый пакет с указанным именем. Как было описано выше, аргументом может быть или символ, или строка. Аргумент `:nicknames` должен быть списком строк, которые будут псевдонимами. И здесь пользователь вместо строк может указывать символы, в случае которых будут использоваться выводимые имена. Это имя и псевдонимы не могут конфликтовать с уже имеющимися именами пакетов. Если конфликт произошёл, сигнализируется ошибка с возможностью исправления.

Аргумент `:use` является списком пакетов или имён (строк или символов) пакетов, чьи внешние символы будут унаследованы новым пакетом. Эти пакеты должны существовать перед вызовом функции.

[Макрос] **in-package** name

Этот макрос пересвязывает переменную `*package*` с пакетом, имя которого указано в параметре *name*. Параметр *name* может быть строкой или символом. Форма *name* не вычисляется. В случае отсутствия пакета, сигнализируется ошибка. Кроме того, в случае вызова в качестве формы верхнего уровня, этот макрос работает и во время компиляции.

`in-package` возвращает новый пакет, то есть значение `*package*` после выполнения операции.

[Функция] **find-package** *name*

Параметр *name* должен быть строкой, которая является именем или псевдонимом искомого пакета. Этот параметр может также быть символов, в случае которого используется выводимое имя. В результате возвращается объект пакета с указанным именем или псевдонимом. Если пакет найден не был, то `find-package` возвращает `nil`. Сравнение имен регистрозависимо (как в `string=`).

Аргумент *package* может быть объектом пакета, в таком случае значение аргумента сразу возвращается. Смотрите раздел 11.2.

[Функция] **package-name** *package*

Аргумент должен быть объектом пакета. Данная функция возвращает строку имени указанного пакета.

Кроме того функция принимает имя пакета или псевдоним, и в этом случае возвращает главное имя пакета. Смотрите раздел 11.2.

`package-name` возвращает `nil` вместо имени пакета, если пакет был удалён. Смотрите `delete-package`.

[Функция] **package-nicknames** *package*

Аргумент должен быть объектом пакета. Эта функция возвращает список псевдонимов для заданного пакета, не включая главное имя.

Кроме того функция принимает имя пакета или псевдоним, и в этом случае возвращается список псевдонимов пакета. Смотрите раздел 11.2.

[Функция] **rename-package** *package new-name* **&optional** *new-nicknames*

Старое имя и все старые псевдонимы пакета *package* удаляются и заменяются на *new-name* и *new-nicknames*. Аргумент *new-name* может быть строкой или символом. Аргумент *new-nicknames*, который по-умолчанию `nil`, является списком строк или символов.

Аргумент *package* может быть как объектом пакета, так и его именем. Смотрите раздел 11.2.

`rename-package` возвращает *package*.

[Функция] **package-use-list** *package*

Данная функция возвращает список пакетов, используемых указанным в параметре пакетом.

Аргумент *package* может быть как объектом пакета, так и его именем. Смотрите раздел 11.2.

[Функция] **package-used-by-list** *package*

Данная функция возвращает список пакетов, использующих указанный в параметре пакет.

Аргумент *package* может быть как объектом пакета, так и его именем. Смотрите раздел 11.2.

[Функция] **package-shadowing-symbols** *package*

Данная функция возвращает список символов, которые были задекларированы, как скрывающие символы, с помощью `shadow` или `shadowing-import`. Все символы в этом списке является *родственными* указанному пакету.

Аргумент *package* может быть как объектом пакета, так и его именем. Смотрите раздел 11.2.

[Функция] **list-all-packages**

Эта функция возвращает список всех пакетов, которые существуют в Lisp'овой системе.

[Функция] **delete-package** *package*

Функция `delete-package` удаляет указанный в аргументе *package* пакет, из всех структур данных системы пакетов. Аргумент *package* может быть как пакетом, так и его именем.

Если *package* является именем, но пакета с данным именем не существует, сигнализируется исправимая ошибка. Продолжение из этой ошибки не делает попытку удаления, но возвращает `nil` из вызова `delete-package`.

Если *package* является пакетов, но пакет уже был удалён, ошибка не сигнализируется и попыток удаления не происходит. Вместо этого `delete-package` немедленно возвращает `nil`.

Если пакет, указанный для удаления, в этот момент используется другими пакетами, сигнализируется исправимая ошибка. Продолжение из этой ошибки выглядит как, если бы функция `unuse-package` была выполнена на всех использующих пакетах для удаления зависимостей, после чего была вызвана `delete-package` для удаления пакета.

Если указанный пакет является домашним для некоторого символа, тогда после выполнения `delete-package`, ячейка домашнего пакета символа становится неопределённой. Никак по-другому символы в удаляемом пакете не модифицируются.

Имя и псевдонимы пакета *package* перестают быть сами собой. Объект пакета остаётся, но анонимно: `packagep` будет истинным, но `package-name` применительно к нему будет возвращать `nil`.

Влияние любой другой операции на удалённым объектом пакета не определена. В частности, попытка найти символ в удалённый пакет (например, используя `intern` или `find-symbol`) будет иметь неопределённый результат.

Если удаление было успешно, `delete-package` возвращает `t`, иначе `nil`.

[Функция] **intern** *string* **&optional** *package*

В пакете *package*, который по-умолчанию равен текущему пакету, осуществляется поиск символа с именем, указанным в аргументе *string*. Этот поиск будет включить унаследованные символы, как описано в разделе 11.4. Если ни одного символа не найдено, то символ создаётся и устанавливается в указанный пакет в качестве внутреннего символа (или внешнего символа, если пакет `keyword`). Указанный пакет

становиться домашним пакетом для созданного символа. Возвращается два значения. Первое является символом, который был найден или создан. Второе значение `nil`, если символ не существовал ранее, или одно из следующих значений:

`:internal` Символ напрямую представлен в пакете как внутренний символ.

`:external` Символ напрямую представлен в пакете как внешний символ.

`:inherited` Символ унаследован с помощью `use-package` (и следовательно символ является внутренним).

Аргумент *package* может быть как объектом пакета, так и его именем. Смотрите раздел 11.2.

[Функция] **find-symbol** *string* **&optional** *package*

Эта функция идентична `intern`, но она никогда не создаёт новый символ. Если символ с указанным именем был найден в указанном пакете, напрямую или через наследование, найденный возвращается символ как первое значение и второе значение так же как в `intern`. Если символ не доступен в заданном пакете, оба значения равны `nil`.

Аргумент *package* может быть как объектом пакета, так и его именем. Смотрите раздел 11.2.

[Функция] **unintern** *symbol* **&optional** *package*

Если указанный символ представлен в указанном пакете *package*, он удаляется из этого пакета и также из списка затеняющих символов, если он там был. Кроме того, если *package* для символа является домашним пакетом, то символ отсоединяется и становится бездомным. Следует отметить, что в некоторых обстоятельствах может продолжить быть доступным с помощью наследования в указанном пакете. Если удаление действительно произошло, `unintern` возвращает `t`, иначе `nil`.

`unintern` должна использоваться с осторожностью. Она изменяет состояние системы пакетов так, что правила согласованности перестают действовать.

Аргумент *package* может быть как объектом пакета, так и его именем. Смотрите раздел 11.2.

[Функция] **export** *symbols* **&optional** *package*

Аргумент *symbols* должен быть списком символов или одиночным символом. Эти символы становятся доступными как внешние символы в пакете *package* (смотрите раздел 11.4). `export` возвращает `t`.

По соглашению, вызов `export` содержит все экспортируемые символы и помещается рядом с началом файла, для указания того, какие символы экспортируются данным файлом для использования другими программами.

Аргумент *package* может быть как объектом пакета, так и его именем. Смотрите раздел 11.2.

[Функция] **unexport** *symbols* **&optional** *package*

Аргумент *symbols* должен быть списком символов или одиночным символом. Эти символы становятся внутренними символами в пакете *package*. Использование этой операции для символов пакета `keyword` является ошибкой. (смотрите раздел 11.4). `unexport` возвращает `t`.

Аргумент *package* может быть как объектом пакета, так и его именем. Смотрите раздел 11.2.

[Функция] **import** *symbols* **&optional** *package*

Аргумент *symbols* должен быть списком символов или одиночным символом. Эти символы становятся внутренними символами в пакете *package* и, таким образом, могут быть указаны без использования полного имени (с двоеточием). `import` сигнализирует исправимую ошибку, если любой из импортируемых символов имеет такое же имя, как один из уже присутствующих в пакете символов (смотрите раздел 11.4). `import` возвращает `t`.

Если любой импортируемый символ не имеет домашнего пакета, тогда `import` устанавливает домашний пакет тот, в который символ импортируется.

Аргумент *package* может быть как объектом пакета, так и его именем. Смотрите раздел 11.2.

[Функция] **shadowing-import** *symbols* **&optional** *package*

Данная функция похожа на `import`, но не сигнализирует ошибку, даже если импортирование символа будет затенять некоторый другой

символ, уже присутствующий в пакете. В дополнение к импортированию символ помещается в список затеняющих символов пакета *package* (смотрите раздел 11.5). `shadowing-import` возвращает `t`.

`shadowing-import` должен использоваться с осторожностью. Она изменяет состояние системы пакетов так, что правила согласованности перестают действовать.

Аргумент *package* может быть как объектом пакета, так и его именем. Смотрите раздел 11.2.

[Функция] **shadow** *symbols* **&optional** *package*

Аргумент должен быть списком символов или строк или одиночным символом или строкой. Рассматривается выводимое имя каждого символа (или просто строка) и в указанном пакете *package* ищется символ с таким же именем. Если данный символ присутствует в этом пакете (напрямую, не с помощью наследования), тогда он добавляется в список затеняющих символов. Иначе, создаётся новый символ с указанным выводимым именем, и вставляется в пакет *package* как внутренний символ. Символ также помещается в список затеняющих символов пакета *package* (смотрите раздел 11.5). `shadow` возвращает `t`.

`shadowing` должен использоваться с осторожностью. Она изменяет состояние системы пакетов так, что правила согласованности перестают действовать.

Аргумент *package* может быть как объектом пакета, так и его именем. Смотрите раздел 11.2.

[Функция] **use-package** *packages-to-use* **&optional** *package*

Аргумент *packages-to-use* должен быть списком пакетов или их имён или одиночным пакетом или его именем. Эти пакеты добавляются в список использования *package*, если их там ещё нет. Все внешние символы в пакетах становятся доступными в пакете *package* как внутренние символы (смотрите раздел 11.4). Использование пакета `keyword` является ошибкой. `use-package` возвращает `t`.

Аргумент *package* может быть как объектом пакета, так и его именем. Смотрите раздел 11.2.

[Функция] **unuse-package** *packages-to-unuse* **&optional** *package*

Аргумент *packages-to-use* должен быть списком пакетов или их имён или одиночным пакетом или его именем. Эти пакеты удаляются из списка использования *package*, если они там были. `unuse-package` возвращает `t`.

Аргумент *package* может быть как объектом пакета, так и его именем. Смотрите раздел 11.2.

[Макрос] **defpackage** defined-package-name {option}*

Эта функция создаёт новый пакет, или модифицирует уже существующий, имя которого является *defined-package-name*. Аргумент *define-package-name* может быть строкой или символом. Если он является символом, то используется его выводимое имя, а не пакет, если вдруг в котором находится данный символ. Созданный или модифицированный пакет возвращается как значение формы `defpackage`.

`(:size` *`integer`*`)` Запись указывает примерное количество символов ожидаемых быть в пакете. Это просто подсказка для менеджера памяти так, что реализация используя хеш-таблицы для хранения структуры пакета может указать данный параметр, и память будет более эффективно выделяться (удобно, например при чтении файла и всех его символов в пакет).

`(:nicknames {}* package-name)` Запись указывает псевдонимы для пакета. Если один из псевдонимов уже указывает на имеющийся пакет, то сигнализируется исправимая ошибка, как для функции `make-package`.

`(:shadow {}* symbol-name)` Указывает имена символов, которые будут затеняющими в определяемом пакете. Работает как `shadow`.

`(:shadowing-import-from` *`package-name`* `{}* symbol-name)`
Указывает имена символов, которые будут размещены в определяемом пакете. Эти символы импортируются в пакет, будучи при необходимости затеняющими, как при использовании функции `shadowing-import`. Ни в коем случае символы не создаются. Если для любого имени *symbol-name* не существует

символа доступного в пакете с именем *package-name*, то сигнализируется исправимая ошибка.

`(:use {}* package-name)` Указывает определяемому пакету «использовать» (наследовать от) пакеты указанные в этой опции. Работает также, как и `use-package`. Если `:use` не указана, тогда опция имеет неопределённое значение.

`(:import-from` *package-name* `{}* symbol-name)` Указывает символы, которые будут размещены в определяемом пакете. Эти символы импортируются в пакет, как при использовании функции `shadowing-import`. Ни в коем случае символы не создаются. Если для любого имени *symbol-name* не существует символа доступного в пакете с именем *package-name*, то сигнализируется исправимая ошибка.

`(:intern {}* symbol-name)` Указанные символы размещаются или создаются в определяемом пакете, как при использовании функции `intern`. Следует отметить, что действие этой опции может зависеть от опции `:use`, так как преимущество отдаётся унаследованному символу, чем созданию нового.

`(:export {}* symbol-name)` Указанные символы размещаются или создаются в определяемом пакете и затем экспортируются, как при использовании функции `export`. Следует отметить, что действие этой опции может зависеть от опции `:use`, `import-from` или `shadowing-import-from` так как преимущество отдаётся унаследованному или импортированному символу, чем созданию нового.

Порядок, в котором опции используются в форме `defpackage`, не имеет значения. Часть соглашения в том, что `defpackage` сортирует опции перед обработкой. Опции обрабатываются в следующем порядке:

1. `:shadow` and `:shadowing-import-from`
2. `:use`
3. `:import-from` and `:intern`
4. `:export`

Сначала устанавливаются затенения, чтобы избежать ложных конфликтов при установке связей для use. Связи use должны

устанавливаться перед импортированием и интернированием, таким образом эти операции могут ссылаться на нормально унаследованные символы, а не создавать новые. Экспортирование выполняется последним, таким образом символы созданные любой другой операцией, в частности, затеняющие и импортированные символы, могут быть экспортированы. Следует отметить, что экспортирование унаследованного символа сначала неявно импортирует его (смотрите раздел 11.4).

Если пакета с именем *defined-package-name* не существовало, то `defpackage` создаст его. Если такой пакет уже существовал, то новый пакет создан не будет. Существующий пакет модифицируется, если возможно, для отражения нового определения. Результат не определён, если новое определение не совместимо с текущим состоянием пакета.

Если одинаковый аргумент `symbol-name` (в смысле сравнения имён с помощью `string=`) встречается более одного раза в опциях `:shadow`, `:shadowing-import-from`, `:import-from` и `:intern`, сигнализируется ошибка.

Если одинаковый аргумент `symbol-name` (в смысле сравнения имён с помощью `string=`) встречается более одного раза в опциях `:intern` и `:export`, сигнализируется ошибка.

Другие виды конфликтов обрабатывается в том же стиле, как в соответствующих операциях `use-package`, `import` и `export`.

Реализация может поддерживать другие опции для `defpackage`. Каждая реализация должно сигнализировать ошибку при встрече с неподдерживаемой опцией для `defpackage`.

Функция `compile-file` должна обрабатывается формы верхнего уровня `defpackage` таким же методом, как и обрабатывает формы верхнего уровня для работы с пакетам (как описано в начале раздела 11.7).

Вот пример вызова `defpackage`, которая «не рискует (plays it safe)», используя только строки в качестве имён.

```
(cl:defpackage "MY-VERY-OWN-PACKAGE"
  (:size 496)
  (:nicknames "MY-PKG" "MYPKG" "MVOP")
  (:use "COMMON-LISP")
```

```
  (:shadow "CAR" "CDR")
  (:shadowing-import-from "BRAND-X-LISP" "CONS")
  (:import-from "BRAND-X-LISP" "GC" "BLINK-FRONT-PANEL-LIGHTS")
  (:export "EQ" "CONS" "MY-VERY-OWN-FUNCTION"))
```

`defpackage` пример выше спроектирован работать корректно, даже если текущий пакет не использует пакет `common-lisp`. (Следует отметить, что в этом примере используется псевдоним `cl` для пакета `common-lisp`.) Более того, выполнение этой формы `defpackage` не создаст символа в текущем пакете. А также для последующего удобства для строк использованы буквы в верхнем регистре.

Заметка для реализации: An implementation of `defpackage` might choose to transform all the *package-name* and *symbol-name* arguments into strings at macro expansion time, rather than at the time the resulting expansion is executed, so that even if source code is expressed in terms of strange symbols in the `defpackage` form, the binary file resulting from compiling the source code would contain only strings. The purpose of this is simply to minimize the creation of useless symbols in production code. This technique is permitted as an implementation strategy but is not a behavior required by the specification of `defpackage`.

А вот для контраста пример `defpackage`, который «играет по крупному (plays the whale)» с использованием всех типов допустимого синтаксиса.

```
(defpackage my-very-own-package
  (:export :EQ common-lisp:cons my-very-own-function)
  (:nicknames "MY-PKG" #:MyPkg)
  (:use "COMMON-LISP")
  (:shadow "CAR")
  (:size 496)
  (:nicknames mvop)
  (:import-from "BRAND-X-LISP" "GC" Blink-Front-Panel-Lights)
  (:shadow common-lisp::cdr)
  (:shadowing-import-from "BRAND-X-LISP" CONS))
```

Этот пример производит то же влияние на вновь созданный пакет, но может создавать бесполезные символы в других пакетах. Использования явных тегов пакетов особенно запутаны. Например, эта форма `defpackage` делает так, что символ `cdr` будет затенён в *новом пакете*. Данный символ не будет затенён в пакете `common-lisp`. Факт, что имя «`CDR`» было указано полным именем (с двоеточием) является отвлекающим маневром. Мораль в том, что синтаксическая гибкость `defpackage`, как в других частях Common Lisp'а, даёт большие удобства в использовании в рамках здравого смысла, но создаёт невообразимую путаницу при использовании мальтузианского изобилия.

Следует отметить, что `defpackage` не способна сама по себе определять взаимно рекурсивные пакеты, например, два пакета использующих друг друга. Однако, ничто не мешает использовать `defpackage` для выполнения первоначальных установок, а затем использовать такие функции, как `use-package`, `import` и `export` для создания связи.

Цель `defpackage` поощрять пользователя размещать определение пакета и его зависимостей в одном месте. Это может также позволить проектировщику большой системы размещать определения всех необходимых пакетов (скажем) в один файл, который может быть загружен перед загрузкой или компиляцией кода, который зависит от этих пакетов. Такой файл, если аккуратно сконструирован, может быть просто загружен в пакет `common-lisp-user`.

Реализации и программные окружения могут также лучше поддерживать процесс разработки (если только предлагая лучшую проверку ошибок) с помощью глобальной базы значений о предполагаемых установках пакетов.

[Функция] **find-all-symbols** *string-or-symbol*

`find-all-symbols` просматривает каждый пакет в Lisp'овой системе для поиска символа, имя которого совпадает с указанной строкой. Список всех найденных символов возвращается в качестве результата. Этот поиск регистрозависимый. Если аргумент является символом, для поиска по строке используется его выводимое имя.

[Макрос] **do-symbols** (var [package [result-form]])
{declaration}* {tag | statement}*

`do-symbols` предоставляет прямой цикл по символам пакета. Тело выполняется единожды для каждого символа доступного в пакете *package*. Символ связывается с переменной *var*. Затем вычисляется форма *result-form* (одиночная форма, *не* неявный `progn`), и результат является результатом формы `do-symbols`. (Когда вычисляется форма *result-form*, переменная *var* продолжает быть связанной и имеет значение `nil`.) Если *result-form* опущена, результат `nil`. Для немедленного завершения цикла может использоваться `return`. Если выполнение тела влияет на то, какие символы содержаться в пакете *package*, то возможно только удаление символа связанного с переменной *var* с использованием `unintern`, иначе результат не определён.

Аргумент *package* может быть как объектом пакета, так и его именем. Смотрите раздел 11.2.

X3J13 voted in March 1988 to specify that the body of a `do-symbols` form may be executed more than once for the same accessible symbol, and users should take care to allow for this possibility.

The point is that the same symbol might be accessible via more than one chain of inheritance, and it is implementationally costly to eliminate such duplicates. Here is an example:

```
(setq *a* (make-package 'a))      ;Implicitly uses package common-lisp
(setq *b* (make-package 'b))      ;Implicitly uses package common-lisp
(setq *c* (make-package 'c :use '(a b)))

(do-symbols (x *c*) (print x))    ;Symbols in package common-lisp
                                  ; might be printed once or twice here
```

X3J13 voted in January 1989 to restrict user side effects; see section 7.9. Следует отметить, что конструкция `loop` предоставляет тип выражения `for`, которое может проходить в цикле по символам пакета (смотрите главу 25).

[Макрос] **do-external-symbols** (var [package [result]])
{declaration}* {tag | statement}*

`do-external-symbols` похожа на `do-symbols` за исключением того,

что сканируются только внешние символы пакета.

The clarification voted by X3J13 in March 1988 for `do-symbols` , regarding redundant executions of the body for the same symbol, applies also to `do-external-symbols`.

Аргумент *package* может быть как объектом пакета, так и его именем. Смотрите раздел 11.2.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

[Макрос] **do-all-symbols** (var [result-form])
{declaration}* {tag | statement}*

Эта функция похожа на `do-symbols`, но выполняет тело единожды для каждого символа каждого пакета. (Функция не обрабатывает все возможные символы, так как символ может быть доступен ни в одном пакете. Обычно, дезинтернированые символы не доступны ни в одном пакете.) В общем случае тело может выполняться для одно символа несколько раз, так как этот символ может встречаться в нескольких пакетах.

The clarification voted by X3J13 in March 1988 for `do-symbols` , regarding redundant executions of the body for the same symbol, applies also to `do-all-symbols`.

Аргумент *package* может быть как объектом пакета, так и его именем. Смотрите раздел 11.2.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

[Макрос] **with-package-iterator** (mname package-list {symbol-type}+)
{form}*

The name *mname* is bound and defined as if by `macrolet`, with the body *form*s as its lexical scope, to be a "generator macro" such that each invocation of `(`*`mname`*`)` will return a symbol and that successive invocations will eventually deliver, one by one, all the symbols from the packages that are elements of the list that is the value of the expression *package-list* (which is evaluated exactly once).

Each element of the *package-list* value may be either a package or the name of a package. As a further convenience, if the *package-list* value is itself a package or the name of a package, it is treated as if a singleton list containing that value had been provided. If the *package-list* value is `nil`, it is considered to be an empty list of packages.

At each invocation of the generator macro, there are two possibilities. If there is yet another unprocessed symbol, then four values are returned: `t`, the symbol, a keyword indicating the accessibility of the symbol within the package (see below), and the package from which the symbol was accessed. If there are no more unprocessed symbols in the list of packages, then one value is returned: `nil`.

When the generator macro returns a symbol as its second value, the fourth value is always one of the packages present or named in the *package-list* value, and the third value is a keyword indicating accessibility: `:internal` means present in the package and not exported; `:external` means present and exported; and `:inherited` means not present (thus not shadowed) but inherited from some package used by the package that is the fourth value.

Each *symbol-type* in an invocation of `with-package-iterator` is not evaluated. More than one may be present; their order does not matter. They indicate the accessibility types of interest. A symbol is not returned by the generator macro unless its actual accessibility matches one of the *symbol-type* indicators. The standard *symbol-type* indicators are `:internal`, `:external`, and `:inherited`, but implementations are permitted to extend the syntax of `with-package-iterator` by recognizing additional symbol accessibility types. An error is signaled if no *symbol-type* is supplied, or if any supplied *symbol-type* is not recognized by the implementation.

The order in which symbols are produced by successive invocations of the generator macro is not necessarily correlated in any way with the order of the packages in the *package-list*. When more than one package is in the *package-list*, symbols accessible from more than one package may be produced once or more than once. Even when only one package is specified, symbols inherited in multiple ways via used packages may be produced once or more than once.

The implicit interior state of the iteration over the list of packages and the symbols within them has dynamic extent. It is an error to invoke the generator macro once the `with-package-iterator` form has been exited.

Any number of invocations of `with-package-iterator` and related macros may be nested, and the generator macro of an outer invocation may be called from within an inner invocation (provided, of course, that its name is visible or otherwise made available).

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

Обоснование: This facility is a bit more flexible in some ways than `do-symbols` and friends. In particular, it makes it possible to implement `loop` clauses for

iterating over packages in a way that is both portable and efficient (see chapter 25).

Глава 12

Числа

Common Lisp содержит несколько различных представлений для чисел. Эти представления могут быть разделены на четыре категории: целые, дробные, с плавающей точкой и комплексные. Большинство функций принимают любой тип числа. Некоторые другие — только определённый тип.

В целом числа в Common Lisp'е не является объектами в прямом понимании слова. Нельзя полагаться, например, на использование `eq` для чисел. В частности, возможно, что значение выражения

```
(let ((x z) (y z)) (eq x y))
```

может быть ложным, а не истинным, если значение `z` является числом.

Обоснование: This odd breakdown of `eq` in the case of numbers allows the implementor enough design freedom to produce exceptionally efficient numerical code on conventional architectures. MacLisp requires this freedom, for example, in order to produce compiled numerical code equal in speed to Fortran. Common Lisp makes this same restriction, if not for this freedom, then at least for the sake of compatibility.

Если два объекта сравниваются для «идентичности», но один из может быть числом, то использование предиката `eql` является более подходящим. Если оба объекта являются числами, то преимущество стоит отдать `=`.

12.1 Точность, неявное приведение и явное приведение

Вычисления с числами с плавающей точкой являются приблизительными. *Точность* чисел с плавающей точкой не обязательно коррелирует с *«аккуратностью»* числа. Например, 3.142857142857142857 имеет более точное приближение к *pi* чем 3.14159, но последнее число более «аккуратно». Точность указывает на количество бит используемых при представлении числа. Если операция объединяла числа с плавающей точкой короткого формата и длинного, то результат будет иметь длинный формат. Это правило создано для уверенности, что при вычислениях аккуратности будет как можно больше. Однако это не гарантированно. Однако, численные процедуры Common Lisp'а предполагают, что аккуратность аргумента не превышает его точность. Поэтому, когда объединяются два числа с плавающей точкой небольшой точности, результатом всегда будет число с плавающей точкой небольшой точности. Это предположение может быть изменено первым же явным преобразованием число с плавающей точкой в более точное представление. (Common Lisp никогда не преобразует числа из более точного формата в менее точный.)

Вычисления рациональных чисел не может вызвать переполнения в обычном смысле этого слова (хотя, конечно, может быть возникнуть недостаток места для представления), целые и дробные числа в принципе могут быть любой величины. Вычисления с плавающей точкой могут вызвать переполнение экспоненты. Это является ошибкой.

X3J13 voted in June 1989 to address certain problems relating to floating-point overflow and underflow, but certain parts of the proposed solution were not adopted, namely to add the macro `without-floating-underflow-traps` to the language and to require certain behavior of floating-point overflow and underflow. The committee agreed that this area of the language requires more discussion before a solution is standardized.

For the record, the proposal that was considered and rejected (for the nonce) introduced a macro `without-floating-underflow-traps` that would execute its body in such a way that, within its dynamic extent, a floating-point underflow must not signal an error but instead must produce either a denormalized number or zero as the result. The rejected proposal

also specified the following treatment of overflow and underflow:

- A floating-point computation that overflows should signal an error of type `floating-point-overflow`.
- Unless the dynamic extent of a use of `without-floating-underflow-traps`, a floating-point computation that underflows should signal an error of type `floating-point-underflow`. A result that can be represented only in denormalized form must be considered an underflow in implementations that support denormalized floating-point numbers.

These points refer to conditions `floating-point-overflow` and `floating-point-underflow` that were approved by X3J13 and are described in section 28.5.

Когда числовой функцией между собой сравниваются или объединяются рациональное и число с плавающей точкой, то вступает в силу правило *неявного приведения к плавающей точке*. Когда рациональное встречает число с плавающей точкой, то рациональное преобразуется в тот же формат второго числа. Для функций, например `+`, которые принимает более двух аргументов, может быть, что все рациональные будут вычислены и результат будет преобразован в число с плавающей точкой.

X3J13 voted in January 1989 to apply the rule of floating-point contagion stated above to the case of *combining* rational and floating-point numbers. For *comparing*, the following rule is to be used instead: When a rational number and a floating-point number are to be compared by a numerical function, in effect the floating-point number is first converted to a rational number as if by the function `rational`, and then an exact comparison of two rational numbers is performed. It is of course valid to use a more efficient implementation than actually calling the function `rational`, as long as the result of the comparison is the same. In the case of complex numbers, the real and imaginary parts are handled separately.

Обоснование: In general, accuracy cannot be preserved in combining operations, but it can be preserved in comparisons, and preserving it makes that part of Common Lisp algebraically a bit more tractable. In particular, this change prevents the breakdown of transitivity. Let `a` be the result of `(/ 10.0 single-float-epsilon)`, and let `j` be the result of `(floor a)`. (Note that `(= a (+ a 1.0))` is true, by the definition of `single-float-epsilon`.) Under the

old rules, all of `(<= a j)`, `(< j (+ j 1))`, and `(<= (+ j 1) a)` would be true; transitivity would then imply that `(< a a)` ought to be true, but of course it is false, and therefore transitivity fails. Under the new rule, however, `(<= (+ j 1) a)` is false.

Для функций, которые математически ассоциативны (и возможно коммутативны), реализация Common Lisp'а может обрабатывать аргументы любым подходящим методом с ассоциативной (и возможно коммутативной) перестановкой. Это конечно не влияет на порядок вычисления форм, данный порядок всегда слева направо, как и во всех Common Lisp'овых вызовах функций. Порядок, который может быть изменён, это обработка значений аргументов. Смысл всего этого в том, что реализация может отличаться в том, какие автоматические приведения типов и в каком порядке производятся. Например, рассмотрим выражение:

(+ 1/3 2/3 1.0D0 1.0 1.0E-15)

Одна реализация может обрабатывать аргументы слева направо, складывая сначала `1/3` и `2/3` для получения `1`, затем преобразовывая результат в число с плавающей точкой двойной точности для сложения с `1.0D0`, затем преобразовывая и добавляя `1.0` и `1.0E-15`. Другая реализация может обрабатывать значения аргументов справа налево, сначала выполняя сложение чисел с плавающей точкой `1.0` и `1.0E-15` (и возможно теряя аккуратность в процессе!), затем преобразовывая результат в число двойной точности и прибавляя `1.0D0`, затем преобразовывая `2/3` к числу с плавающей точкой двойной точности, и затем преобразовывая `1/3` и добавляя его. Третья реализация может сначала просканировать все аргументы, и сгруппировав их по типам, выполнить сложения сначала одинаковых типов, затем преобразовать результаты к наиболее точному типу и сложить их. В этом случае все три стратегии являются допустимыми. Пользователь конечно может контролировать порядок обработки аргументов явно задавая вызовы вычислений, например:

(+ (+ 1/3 2/3) (+ 1.0D0 1.0E-15) 1.0)

Пользователь может также контролировать приведения, явно используя для этого функцию.

В целом, тип результата числовой функции является числом с плавающей точкой наиболее точного формата, который был в аргументах данной функции. Но если все аргументы были рационального типа, тогда результат будет рациональным (за исключением функций, который математически возвращают иррациональные результаты, в случае который используется одинарная точность с плавающей точкой)

Другое правило для комплексных чисел. Комплексные числа никогда не возвращаются из числовых функций, если только в аргументах не было использовано хоть одно комплексное число. (Исключением из этого правила являются иррациональные и трансцендентальные функции, в частности `expt`, `log`, `sqrt`, `asin`, `acos`, `acosh` и `atanh`. Смотрите раздел 12.5.) Когда некомплексное число встречает комплексно, то первое сначала конвертируется во второе с нулевой мнимой частью, а потом вычисляется результат.

Если любое вычисление привело к дробному результату, в котором числитель нацело делится на знаменатель, то результат немедленно преобразуется к эквивалентному целому числу. Это правило называется *канонизацией дробей*.

Если результат любого вычисления должен быть комплексным числом с рациональными частями и нулевой мнимой частью, то результат немедленно преобразуется в некомплексное рациональное число и равен действительной части исходного. Это называется правилом *канонизации комплексного числа*. Следует отметить, что это правило *не* применяется к комплексным числам с компонентами из чисел с плавающими точками. Таким образом `#C(5 0)` и `5` равны `eql`, а `#C(5.0 0.0)` и `5.0` не равны `eql`, но равны `equalp`.

12.2 Предикаты для чисел

Каждая из следующих функций проверяет одиночное число на наличие некоторого свойства. Каждая функция требует числовой аргумент. Вызов с аргументом другого типа является ошибкой.

[Функция] **zerop** *number*

Если *number* является нулём (целым нулём, нулём с плавающей точкой, комплексным нулём), этот предикат истинен, иначе ложен. Вне зависимости от того, предоставляет различные представления для положительного и отрицательного нулей система, `(zerop -0.0)` всегда истинно. Если аргумент *number* не является числом, то сигнализируется ошибка.

[Функция] **plusp** *number*

Если *number* строго больше нуля, то предикат истинен, иначе ложен. Аргумент *number* должен быть любым числом, кроме комплексного, иначе сигнализируется ошибка.

[Функция] **minusp** *number*

Если *number* строго меньше нуля, то предикат истинен, иначе ложен. Вне зависимости от того, предоставляет ли система различные отображения для отрицательного и положительного нулей с плавающей точкой, `(minusp -0.0)` всегда ложно. (Для проверки отрицательного нуля, может использоваться функция `float-sign`.) Аргумент *number* должен быть любым числом, кроме комплексного, иначе сигнализируется ошибка.

[Функция] **oddp** *integer*

Если аргумент *integer* является нечётным числом (то есть не делится на два нацело), то предикат истинен, иначе ложен. Если аргумент не целое число, сигнализируется ошибка.

[Функция] **evenp** *integer*

Если аргумент *integer* является чётным числом (то есть делится на два нацело), то предикат истинен, иначе ложен. Если аргумент не целое число, сигнализируется ошибка.

Смотрите также предикаты типов данных `integerp`, `rationalp`, `floatp`, `complexp` и `numberp`.

12.3 Сравнение чисел

Каждая функция из данного раздела требует, чтобы все аргументы были числами. Использование в качестве аргумента не числа является ошибкой. Если не указано иное, то каждая функция работает со всеми типами чисел, автоматически выполняя необходимые приведения, когда типы аргументы различаются.

[Функция] = *number* **&rest** *more-numbers*
[Функция] /= *number* **&rest** *more-numbers*
[Функция] < *number* **&rest** *more-numbers*
[Функция] > *number* **&rest** *more-numbers*
[Функция] <= *number* **&rest** *more-numbers*
[Функция] >= *number* **&rest** *more-numbers*

Каждая из этих функций принимает один и более аргументов. Если последовательно аргументов удовлетворяет следующему условию:

`=`	все равны
`/=`	все различны
`<`	монотонно возрастают
`>`	монотонно убывают
`<=`	монотонно не убывают
`>=`	монотонно не возрастают

тогда предикат истинен, иначе ложен. Комплексные числа могут сравниваться с помощью `=` и `/=`, но остальные предикаты требуют некомплексных аргументов. Два комплексных числа равны `=`, если их действительные части равны между собой и мнимые равны между собой с помощью предиката `=`. Комплексное число может быть сравнено с некомплексным с помощью `=` или `/=`. Например

(= 3 3) истина	(/= 3 3) ложь
(= 3 5) ложь	(/= 3 5) истина
(= 3 3 3 3) истина	(/= 3 3 3 3) ложь
(= 3 3 5 3) ложь	(/= 3 3 5 3) ложь
(= 3 6 5 2) ложь	(/= 3 6 5 2) истина
(= 3 2 3) ложь	(/= 3 2 3) ложь

(< 3 5) истина
(< 3 -5) ложь
(< 3 3) ложь
(< 0 3 4 6 7) истина
(< 0 3 4 4 6) ложь
(> 4 3) истина
(> 4 3 2 1 0) истина
(> 4 3 3 2 0) ложь
(> 4 3 1 2 0) ложь
(= 3) истина
(< 3) истина
(= 3.0 #C(3.0 0.0)) истина
(= 3 3.0) истина
(= 0.0 -0.0) истина
(> 0.0 -0.0) ложь

(<= 3 5) истина
(<= 3 -5) ложь
(<= 3 3) истина
(<= 0 3 4 6 7) истина
(<= 0 3 4 4 6) истина
(>= 4 3) истина
(>= 4 3 2 1 0) истина
(>= 4 3 3 2 0) истина
(>= 4 3 1 2 0) ложь
(/= 3) истина
(<= 3) истина
(/= 3.0 #C(3.0 1.0)) истина
(= 3.0s0 3.0d0) истина
(= 5/2 2.5) истина
(= 0 -0.0) истина

С двумя аргументами, эти функции выполняют обычные арифметические сравнения. С тремя и более аргументами, они полезны для проверок рядов, как показано в следующем примере:

```
(<= 0 x 9)                  ;истина, если x между 0 и 9, включительно
(< 0.0 x 1.0)               ;истина, если x между 0.0 и 1.0
(< -1 j (length s))         ;истина, если j корректный индекс для s
(<= 0 j k (- (length s) 1))     ;истина, если j и k оба корректные
                            ;индексы s и j ≤ k
```

Обоснование: Отношение «неравенства» называется `/=`, а не `<>` (как в Pascal'е) по двум причинам. Первое, `/=` для более чем двух аргументов не означает то же, что и `или <`, или `>`. Второе, неравенство для комплексная не означает то же, что и `<` и `>`. По этим двум причинам неравенство не означает больше или меньше.

[Функция] **max** *number* **&rest** *more-numbers*
[Функция] **min** *number* **&rest** *more-numbers*

Аргументы могут быть некомплексными числами. `max` возвращает наибольший аргумент (ближайший к положительной бесконечности).

`max` возвращает наименьший аргумент (ближайший к отрицательной бесконечности).

Для `max`, если аргументы являются смесью из дробных и с плавающей точкой чисел, и наибольший является дробным, реализация может вернуть как дробное число, так и его аппроксимацию с плавающей точкой. Если наибольший аргумент является числом с плавающей точкой меньшего формата в сравнении с числом большего формата, то реализация может вернуть число без изменения, либо конвертировать его в больший формат. More concisely, the implementation has the choice of returning the largest argument as is or applying the rules of floating-point contagion, taking all the arguments into consideration for contagion purposes. Если среди наибольших два аргумента равны, то реализация может вернуть любой из них. Те же правила применяются и ко второй функции `min` (только «наибольший аргумент» нужно заменить на «наименьший аргумент»).

(max 6 12) ⇒ 12
(max -6 -12) ⇒ -6
(max 1 3 2 -7) ⇒ 3
(max -2 3 0 7) ⇒ 7
(max 3) ⇒ 3
(max 5.0 2) ⇒ 5.0
(max 3.0 7 1) ⇒ 7 *или* 7.0
(min 6 12) ⇒ 6
(min -6 -12) ⇒ -12
(min 1 3 2 -7) ⇒ -7
(min -2 3 0 7) ⇒ -2
(min 3) ⇒ 3
(min 5.0 2) ⇒ 2 *или* 2.0
(min 3.0 7 1) ⇒ 1 *или* 1.0
(max 1.0s0 7.0d0) ⇒ 7.0d0
(min 1.0s0 7.0d0) ⇒ 1.0s0 *или* 1.0d0
(max 3 1 1.0s0 1.0d0) ⇒ 3 *или* 3.0d0
(min 3 1 1.0s0 1.0d0) ⇒ 1 *или* 1.0s0 *или* 1.0d0

12.4 Арифметические операции

Каждая из функций в этом разделе в качестве аргументов принимает числа. Применение какой-либо из них к нечисловому аргументу является ошибкой. Каждая из них работает со всеми типами чисел,

автоматически приводя типы для различных аргументов, если не указано иное.

[Функция] + **&rest** *numbers*

Данная функция возвращает сумму всех аргументов. Если аргументов не было, результатом является `0`.

[Функция] **-** *number* **&rest** *more-numbers*

Функция `-` при использовании с одним аргументом возвращает отрицательное значение для этого аргумента.

Функция `-` при использовании с более чем одним аргументом вычитает из первого аргумента все остальные и возвращает результат. Например, `(- 3 4 5)` ⇒ `-6`.

[Функция] ***** **&rest** *numbers*

Функция возвращает произведение всех аргументов. Если вызывается без аргументов, тогда возвращает `1`.

[Функция] / *number* **&rest** *more-numbers*

Функция `/` при использовании с более чем одним аргументом производит деление первого аргумента на все остальные значения и возвращает полученный результат.

Ноль может использоваться только для первого аргумента.

Для одного аргумента `/` возвращает обратное значение. В таком случае ноль в первом аргументе недопустим.

Если математическое отношение двух целых чисел не является целым числом, `/` вернёт дробный тип. Например:

(/ 12 4) ⇒ 3
(/ 13 4) ⇒ 13/4
(/ -8) ⇒ -1/8
(/ 3 4 5) ⇒ 3/20

Для целочисленного деления используйте одну из функций `floor`, `ceiling`, `truncate` или `round`.

Если один из аргументов является числом с плавающей точкой, тогда применяются правила обработки чисел с плавающей точкой.

[Функция] **1+** *number*
[Функция] **1-** *number*

`(1+ x)` является тем же самым, что `(+ x 1)`. `(1- x)` является тем же самым, что `(- x 1)`. Следует отметить, что короткое имя может ввести в заблуждение: `(1- x)` *не значит* то же, что и $1 - x$.

Заметка для реализации: Compiler writers are very strongly encouraged to ensure that `(1+ x)` and `(+ x 1)` compile into identical code, and similarly for `(1- x)` and `(- x 1)`, to avoid pressure on a Lisp programmer to write possibly less clear code for the sake of efficiency. This can easily be done as a source-language transformation.

[Макрос] **incf** place [delta]
[Макрос] **decf** place [delta]

Значение обобщённой переменной *place* увеличивается (`incf`) или уменьшается (`decf`) на значение формы *delta* и затем присваивается переменной *place*. Результатом формы является присвоенное значение. Форма *place* может быть любой формой подходящей для `setf`. Если форма *delta* не указана, тогда число в *place* изменяется на `1`. Например:

```
(setq n 0)
(incf n) ⇒ 1          and now n ⇒ 1
(decf n 3) ⇒ -2       and now n ⇒ -2
(decf n -5) ⇒ 3       and now n ⇒ 3
(decf n) ⇒ 2          and now n ⇒ 2
```

Результат `(incf` *`place delta`*`)` примерно эквивалентен

(setf *place* (+ *place delta*))

за исключением того, что в последнем случае *place* вычисляется дважды, тогда как `incf` гарантирует, что вычисление *place* будет выполнено только один раз. Более того, для некоторых форма *place* `incf` может быть эффективнее чем `setf`. X3J13 voted in March 1988 to clarify order of evaluation (see section 7.2).

[Функция] **conjugate** *number*

Функция возвращает комплексное сопряжение с числа *number*. Сопряжением некомплексного числа является само число. Для комплексного числа `z`,

(conjugate z) ≡ (complex (realpart z) (- (imagpart z)))

Например,

(conjugate #C(3/5 4/5)) ⇒ #C(3/5 -4/5)
(conjugate #C(0.0D0 -1.0D0)) ⇒ #C(0.0D0 1.0D0)
(conjugate 3.7) ⇒ 3.7

[Функция] **gcd** **&rest** *integers*

Функция возвращает наибольший общий делитель всех аргументов, которые в свою очередь должны быть целыми числами. Результатом `gcd` всегда является неотрицательное целое число. Если передан только один параметр, то возвращается его абсолютное значение (модуль). Если ни один параметр не был передан, `gcd` возвращает `0`. Для трёх и более аргументов,

(gcd *a b c ... z*) ≡ (gcd (gcd *a b*) *c ... z*)

Несколько примеров использования `gcd`:

(gcd 91 -49) ⇒ 7
(gcd 63 -42 35) ⇒ 7
(gcd 5) ⇒ 5
(gcd -4) ⇒ 4
(gcd) ⇒ 0

[Функция] **lcm** *integer* **&rest** *more-integers*

Функция возвращает наименьшее общее кратное для аргументов, которые в свою очередь должны быть целыми числами. Результатом `lcm` всегда является неотрицательное целое число. Для двух аргументов, не являющихся нулями,

(lcm a b) $\equiv$ (/ (abs (* a b)) (gcd a b))

Если один или оба аргумента является нулём.

(lcm a 0) $\equiv$ (lcm 0 a) $\equiv$ 0

Для одного аргумента, `lcm` возвращает абсолютное значение этого аргумента (модуль). Для трёх и более аргументов,

(lcm a b c ... z) $\equiv$ (lcm (lcm a b) c ... z)

Примеры использования:

(lcm 14 35) $\Rightarrow$ 70
(lcm 0 5) $\Rightarrow$ 0
(lcm 1 2 3 4 5 6) $\Rightarrow$ 60

`lcm`, вызванная без аргументов, возвращает `1`.

(lcm) $\Rightarrow$ 1

12.5 Иррациональные и трансцендентные функции

Common Lisp не содержит тип данных, который точно отображает иррациональные числовые значения. В данном разделе функции описаны так, как если бы результаты были математически точными, но фактически все они возвращают число с плавающей точкой приблизительно равное настоящему математическому значению. В некоторых местах изложены математические тождества, связанные со значениями функций, однако, два математически идентичных выражения могут быть различными по причине ошибок процесса аппроксимации при вычислениях чисел с плавающей точкой.

Когда все аргументы для функции данного раздела являются рациональными и Ъ математический результат также является (математически) рациональным, тогда, если не указано иное, реализация может вернуть или точный результат типа `rational` или приближенное значение с плавающей точкой одинарной точности. Если все аргументы рациональны, но результат не может быть рациональным, тогда возвращается число с плавающей точкой одинарной точности.

Если все переданные в функцию аргументы принадлежат типу `(or rational (complex rational))` и Ъ математический результат является (математически) комплексным числом с рациональными действительной и мнимой частями, тогда, если не указано иное, реализация может вернуть или точный результат типа `(or rational (complex rational))` или приближенный типа с плавающей точкой с одинарной точностью `single-float` (только если мнимая часть равна нулю) или `(complex single-float)`. Если все аргументы типа `(or rational (complex rational))`, но результат не может быть выражен рациональными или комплексным рациональным числом, тогда он будет принадлежать типу `single-float` (только если мнимая часть равна нулю) или `(complex single-float)`.

Все функции за исключением `expt` подчиняются правилам неявного приведения плавающей точки или комплексного числа. Когда, возможно после приведения типов, все аргументы становятся с плавающей точкой или комплексным числом с плавающей точкой, тогда результат будет такого же типа, что и аргумента, если не указано иное.

Заметка для реализации: Для понимания работы функций из данного

раздела может быть полезной «поваренная книга чисел с плавающей точкой» от Cody и Waite [14].

12.5.1 Экспоненциальные и логарифмические функции

Наряду с обычными одно-аргументными и двух-аргументными экспоненциальными и логарифмическими функциями, `sqrt` рассматривается как экспоненциальная функция, потому что она возводит число в степень 1/2.

[Функция] **exp** *number*

Возвращает *e*, возведённое в степень *number*, где *e* является основанием натурального логарифма.

[Функция] **expt** *base-number power-number*

Возвращает *base-number*, возведённый в степень *power-number*. Если *base-number* принадлежит типу `rational` и *power-number* — `integer`, тогда результат вычислений будет принадлежать типу `rational`, в противном случае результат будет приближенным числом с плавающей точкой.

X3J13 voted in March 1989 to clarify that provisions similar to those of the previous paragraph apply to complex numbers. If the *base-number* is of type `(complex rational)` and the *power-number* is an `integer`, the calculation will also be exact and the result will be of type `(or rational (complex rational))`; otherwise a floating-point or complex floating-point approximation may result.

Если *power-number* равен `0` (ноль целочисленного типа), тогда результат всегда будет значение 1, такого же типа что и *base-number*, даже если *base-number* равен нулю (любого типа). То есть:

(expt x 0) $\equiv$ (coerce 1 (type-of x))

Если *power-number* является нулём любого другого типа данных, тогда результат также равен 1 такого же типа, что и аргументы после применения правил приведения числовых типов, с одним условием: если *base-number* равен нулю, когда *power-number* является нецелочисленным нулём, то это является ошибкой.

Реализация `expt` может использовать различные алгоритмы для случаев с рациональным и с плавающей точкой аргументом *power-number*. Суть в том, что в большинстве случаев боле аккуратный результат может быть достигнут для рационального *power-number*. Например, `(expt pi 16)` и `(expt pi 16.0)` могут вернуть слегка разные результаты, если в первом случае алгоритм «повторных квадратов», а во втором использование логарифмов.

Результатом `expt` может быть комплексное число, даже если ни один аргумент не был комплексным. Такой результат получается если *base-number* отрицательное число и *power-number* не является целочисленным. Следует отметить, что `(expt -8 1/3)` не может вернуть `-2`, хотя и `-2` несомненно является одним из кубических корней для `-8`, но основным корнем является аппроксимированное комплексное число `#C(1.0 1.73205)`.

[Функция] **log** *number* **&optional** *base*

Функция возвращает логарифм числа *number* с основанием *base*, которое по умолчанию равно *e* (эпсилон, основание для натурального логарифма). Например:

(log 8.0 2) ⇒ 3.0
(log 100.0 10) ⇒ 2.0

В зависимости от реализации, результатом `(log 8 2)` может быть как `3`, так и `3.0`. Например:

(log -1.0) ≡ (complex 0.0 (float pi 0.0))

X3J13 voted in January 1989 to specify certain floating-point behavior when minus zero is supported. As a part of that vote it approved a mathematical definition of complex logarithm in terms of real logarithm, absolute value, arc tangent of two real arguments, and the phase function as

Logarithm $\log|z| + i\,\mathrm{phase}\,z$

This specifies the branch cuts precisely whether minus zero is supported or not; see `phase` and `atan`.

[Функция] **sqrt** *number*

Функция возвращает квадратный корень числа *number*. Если *number* не является комплексным числом и отрицательно, тогда результат будет комплексным числом. Например:

(sqrt 9.0) ⇒ 3.0
(sqrt -9.0) ⇒ #c(0.0 3.0)

В зависимости от реализации результат `(sqrt 9)` может быть как `3`, так и `3.0`. Результат `(sqrt -9)` может быть как `#(0 3)` или `#c(0.0 3.0)`.

X3J13 voted in January 1989 to specify certain floating-point behavior when minus zero is supported. As a part of that vote it approved a mathematical definition of complex square root in terms of complex logarithm and exponential functions as

Square root $e^{(\log z)/2}$

This specifies the branch cuts precisely whether minus zero is supported or not; see `phase` and `atan`.

[Функция] **isqrt** *integer*

Целочисленный квадратный корень: аргумент должен быть неотрицательным целым, и результат является наибольшим целым числом, которое меньше или равно точному положительному квадратному корню аргумента.

(isqrt 9) ⇒ 3
(isqrt 12) ⇒ 3
(isqrt 300) ⇒ 17
(isqrt 325) ⇒ 18

12.5.2 Тригонометрические и связанные с ними функции

Некоторые из функций в данном разделе, такие как `abs` и `signum`, несомненно не относятся к тригонометрическим функциям, когда рассматриваются как функции только для действительных чисел. Однако, путь, с помощью которого они расширены для операций на комплексных числах, делает их связь с тригонометрическими функциями явной.

[Функция] **abs** *number*

Возвращает абсолютное значение аргумента. Для некомплексных чисел x,

(abs x) ≡ (if (minusp x) (- x) x)

и результат всегда имеет тот же тип, что и аргумент.

Для комплексных чисел z, абсолютное значение может быть вычислено как

(sqrt (+ (expt (realpart z) 2) (expt (imagpart z) 2)))

Заметка для реализации: Аккуратные разработчики не будут напрямую использовать эту формулу для всех комплексных чисел. Очень большие и очень маленькие части комплексных чисел будут обрабатываться специализированно для избежания выходов за верхние и нижние границы значений.

Например:

(abs #c(3.0 -4.0)) ⇒ 5.0

Результатом `(abs #(3 4))` может быть или `5` или `5.0`, в зависимости от реализации.

[Функция] **phase** *number*

Аргументом (так называется функция phase) числа z (arg(z) ≡(phase z)) является угол φ (в радианах) радиус-вектора точки, соответствующей комплексному числу z.

X3J13 voted in January 1989 to specify certain floating-point behavior when minus zero is supported; `phase` is still defined in terms of `atan` as above, but thanks to a change in `atan` the range of `phase` becomes $-\pi$ *inclusive* to π inclusive. The value $-\pi$ results from an argument whose real part is negative

and whose imaginary part is minus zero. The `phase` function therefore has a branch cut along the negative real axis. The phase of $+0 + 0i$ is $+0$, of $+0 - 0i$ is -0, of $-0 + 0i$ is $+\pi$, and of $-0 - 0i$ is $-\pi$.

Если аргумент является комплексным числом с частями из чисел с плавающей точкой, то результатом является число с плавающей точкой того же типа. Если аргумент является числом с плавающей точкой, то результатом является число с плавающей точкой того же типа. Если аргумент является комплексным числом с частями из рациональных чисел, то результатом является число с плавающей точкой одинарного типа.

[Функция] **signum** *number*

По определению,

(signum x) $\equiv$ (if (zerop x) x (/ x (abs x)))

Для рационального числа, `signum` будет возвращать один из вариантов: `-1`, `0` или `1`, в соответствии с тем, является ли число отрицательным, нулём или положительным. Для числа с плавающей точкой, результатом будет число с плавающей точкой того же типа и значением: -1, 0 или 1. Для комплексного числа z, `(signum z)` является комплексным числом с таким же аргументом (phase), но с единичным модулем. Но если z равен комплексному нулю, результатом является само число z. Например:

(signum 0) ⇒ 0
(signum -3.7L5) ⇒ -1.0L0
(signum 4/5) ⇒ 1
(signum #C(7.5 10.0)) ⇒ #C(0.6 0.8)
(signum #C(0.0 -14.7)) ⇒ #C(0.0 -1.0)

Для некомплексных рациональных чисел `signum` является рациональной функцией, но для комплексных чисел может быть иррациональной.

[Функция] **sin** *radians*
[Функция] **cos** *radians*
[Функция] **tan** *radians*

`sin` возвращает синус аргумента, `cos` — косинус, `tan` — тангенс. Аргумент рассматривается как угол в радианах. Аргумент может быть комплексным числом.

[Функция] **cis** *radians*

Функция вычисляет $e^{i \cdot radians}$. Имя `cis` обозначает «cos + i sin», потому что $e^{i\theta} = \cos\theta + i\sin\theta$. Аргумент рассматривается как угол в радианах и может быть любым некомплексным числом. Результатом является комплексное число, у которого действительная часть это косинус аргумента, и мнимая — синус аргумента. Другими словами, результат это комплексное число, у которого аргумент (фаза) равна (mod $2/pi$) и модуль равен единице.

Заметка для реализации: Чаще всего дешевле вычислить синус и косинус угла вместе, чем выполнять два раздельных вычисления.

[Функция] **asin** *number*
[Функция] **acos** *number*

`asin` возвращаем арксинус аргумента, и `acos` — арккосинус аргумента. Результат будет в радианах. Аргумент может быть комплексным числом.

Функции арксинус и арккосинус могут быть математически определены для аргумента z следующим образом:

Арксинус $-i\log\left(iz+\sqrt{1-z^2}\right)$
Арккосинус $-i\log\left(z+i\sqrt{1-z^2}\right)$

Следует отметить, что результат `asin` или `acos` может быть комплексным числом, даже если аргумент не являлся комплексным. Такое происходит, когда абсолютное значение аргумента превышает 1.

Kahan [25] suggests for `acos` the defining formula

Arc cosine $\dfrac{2\log\left(\sqrt{\frac{1+z}{2}}+i\sqrt{\frac{1-z}{2}}\right)}{i}$

or even the much simpler $(\pi/2) - \arcsin z$. Both equations are mathematically equivalent to the formula shown above.

Заметка для реализации: These formulae are mathematically correct, assuming completely accurate computation. They may be terrible methods for floating-point computation. Implementors should consult a good text on numerical analysis. The formulae given above are not necessarily the simplest ones for real-valued computations, either; they are chosen to define the branch cuts in desirable ways for the complex case.

[Функция] **atan** *y* **&optional** *x*

Функция вычисляет арктангенс и возвращает результат в радианах.

Ни один из двух аргументов y и x не может быть комплексным. Знаки y и x используются для определения квадранта. x может быть нулём при условии, что y не ноль. Значение `atan` лежит между $-pi$ (невключительно) и pi (включительно). Следующая таблица описывает различные специальные случаи.

Условие	Декартово местоположение	Промежуток результата
$y = +0 x > 0$	Точно выше положительной оси x	$+0$
$y > 0 \quad x > 0$	Квадрант I	$+0 < result < \pi/2$
$y > 0 \quad x = \pm 0$	Положительная ось y	$\pi/2$
$y > 0 \quad x < 0$	Квадрант II	$\pi/2 < result < \pi$
$y = +0 x < 0$	Точно ниже негативной оси x	π
$y = -0 x < 0$	Точно выше отрицательной оси x	π
$y < 0 \quad x < 0$	Квадрант III	$-\pi < result < -\pi/2$
$y < 0 \quad x = \pm 0$	Отрицательная ось y	$-\pi/2$
$y < 0 \quad x > 0$	Квадрант IV	$-\pi/2 < result < -0$
$y = -0 x > 0$	Точно ниже положительной оси x	-0
$y = +0 x = +0$	Рядом с центром	$+0$
$y = -0 x = +0$	Рядом с центром	-0
$y = +0 x = -0$	Рядом с центром	π
$y = -0 x = -0$	Рядом с центром	$-\pi$

Следует отметить, что случай $y = 0, x = 0$ при отсутствии минус ноля является ошибкой, но четыре случая $y = \pm 0, x = \pm 0$ определены при условии существования минус ноля.

Если не указывать x, y может быть комплексным. Результатом функции является арктангенс y, который может быть определён следующей формулой:

Арктангенс $\frac{\log(1+iy)-\log(1-iy)}{2i}$

Для некомплексного аргумента y, результат будет некомплексным и будет лежать между $-\pi/2$ и $\pi/2$ (оба невключительно).

Заметка для реализации: This formula is mathematically correct, assuming completely accurate computation. It may be a terrible method for floating-point computation. Implementors should consult a good text on numerical analysis. The formula given above is not necessarily the simplest one for real-valued computations, either; it is chosen to define the branch cuts in desirable ways for the complex case.

[Константа] **pi**

Данная глобальная переменная содержит значение наиболее приближенное к *pi* в *длинном* формате числа с плавающей точкой. Например:

```
(defun sind (x)     ;Аргумент в градусах
  (sin (* x (/ (float pi x) 180))))
```

Приближение к *pi* с другими точностями может быть выполнено с помощью `(float pi x)`, где x является числом с плавающей точкой необходимой точности, или с помощью `(coerce pi type)`, где *type* является именем необходимого типа, как например `short-float`.

[Функция] **sinh** *number*
[Функция] **cosh** *number*
[Функция] **tanh** *number*
[Функция] **asinh** *number*
[Функция] **acosh** *number*
[Функция] **atanh** *number*

Данные функции вычисляют гиперболический синус, косинус, тангенс, арксинус, арккосинус и арктангенс, которые для аргумента z математически определены следующим образом:

Гиперболический синус $(e^z - e^{-z})/2$
Гиперболический косинус $(e^z + e^{-z})/2$
Гиперболический тангенс $(e^z - e^{-z})/(e^z + e^{-z})$
Гиперболический арксинус $\log\left(z + \sqrt{1 + z^2}\right)$
Гиперболический арккосинус $\log\left(z + (z+1)\sqrt{(z-1)/(z+1)}\right)$
Гиперболический арктангенс $\log\left((1+z)\sqrt{1/(1-z^2)}\right)$

Следует отметить, что результат `acosh` может быть комплексным, даже если аргумент комплексным числом не был. Это происходит если аргумент меньше 1. Также результат `atanh` может быть комплексным, даже если аргумент не был комплексным. Это происходит если абсолютное значение аргумента было больше 1.

Заметка для реализации: Эти формулы математически корректны, предполагая совершенно точные вычисления. Но они могут быть неудобны для вычислений чисел с плавающей точкой. Разработчики реализация должны ознакомится с хорошей книгой по численному анализу. Вышеприведённые формы не обязательно являются самыми простыми для вещественных вычислений. Они выбраны для определения точек ветвлений для случаев использования комплексных чисел.

12.5.3 Точки ветвления, главные значения и краевые условия на комплексной плоскости

Многие иррациональные и трансцендентные функции на комплексной плоскости многозначны. Например, в общем случае, логарифмическая функция имеет бесконечное количество комплексных значений. В каждом таком случае для возврата должно быть выбрано одно главное значение.

Common Lisp определяет точки ветвления, главные значения и краевые условия для комплексных функции в соотвествие с предложениями для комплексных функций в APL [36]. Содержимое раздела по большей части скопировано из этих предложений.

Indeed, X3J13 voted in January 1989 to alter the direction of continuity for the branch cuts of `atan`, and also to address the treatment of branch cuts in implementations that have a distinct floating-point minus zero.

The treatment of minus zero centers in two-argument `atan`. If there is no minus zero, then the branch cut runs just below the negative real axis as before, and the range of two-argument `atan` is $(-\pi, \pi]$. If there is a minus zero, however, then the branch cut runs precisely on the negative real axis, skittering between pairs of numbers of the form $-x \pm 0i$, and the range of two-argument `atan` is $[-\pi, \pi]$.

The treatment of minus zero by all other irrational and transcendental functions is then specified by defining those functions in terms of two-argument `atan`. First, `phase` is defined in terms of two-argument `atan`, and complex `abs` in terms of real `sqrt`; then complex `log` is defined in terms of `phase`, `abs`, and real `log`; then complex `sqrt` in terms of complex `log`; and finally all others are defined in terms of these.

Kahan [25] treats these matters in some detail and also suggests specific algorithms for implementing irrational and transcendental functions in IEEE standard floating-point arithmetic [23].

Remarks in the first edition about the direction of the continuity of branch cuts continue to hold in the absence of minus zero and may be ignored if minus zero is supported; since all branch cuts happen to run along the principal axes, they run *between* plus zero and minus zero, and so each sort of zero is associated with the obvious quadrant.

`sqrt` The branch cut for square root lies along the negative real axis, continuous with quadrant II. The range consists of the right half-plane, including the non-negative imaginary axis and excluding the negative imaginary axis.

X3J13 voted in January 1989 to specify certain floating-point behavior when minus zero is supported. As a part of that vote it approved a mathematical definition of complex square root:

$$\sqrt{z} = e^{(\log z)/2}$$

This defines the branch cuts precisely, whether minus zero is supported or not.

`phase` The branch cut for the phase function lies along the negative real axis, continuous with quadrant II. The range consists of that portion of the real axis between $-\pi$ (exclusive) and π (inclusive).

X3J13 voted in January 1989 to specify certain floating-point behavior when minus zero is supported. As a part of that vote it approved a mathematical definition of phase:

$$\text{phase}\, z = \arctan(\Im z, \Re z)$$

where $\Im z$ is the imaginary part of z and $\Re z$ the real part of z. This defines the branch cuts precisely, whether minus zero is supported or not.

`log` The branch cut for the logarithm function of one argument (natural logarithm) lies along the negative real axis, continuous with quadrant II. The domain excludes the origin. For a complex number z, $\log z$ is defined to be

$$\log z = (\log |z|) + i(\text{phase}\, z)$$

Therefore the range of the one-argument logarithm function is that strip of the complex plane containing numbers with imaginary parts between $-\pi$ (exclusive) and π (inclusive).

The X3J13 vote on minus zero would alter that exclusive bound of $-\pi$ to be inclusive if minus zero is supported.

The two-argument logarithm function is defined as $\log_b z = (\log z)/(\log b)$. This defines the principal values precisely. The range of the two-argument logarithm function is the entire complex plane. It is an error if z is zero. If z is non-zero and b is zero, the logarithm is taken to be zero.

`exp` The simple exponential function has no branch cut.

`expt` The two-argument exponential function is defined as $b^x = e^{x \log b}$. This defines the principal values precisely. The range of the two-argument exponential function is the entire complex plane. Regarded as a function of x, with b fixed, there is no branch cut. Regarded as a function of b, with x fixed, there is in general a branch cut along the negative real axis, continuous with quadrant II. The domain excludes the origin. By definition, $0^0 = 1$. If $b = 0$ and the real part of x is strictly positive, then $b^x = 0$. For all other values of x, 0^x is an error.

`asin` The following definition for arc sine determines the range and branch cuts:

$$\arcsin z = -i \log\left(iz + \sqrt{1 - z^2}\right)$$

This is equivalent to the formula

$$\arcsin z = \frac{\operatorname{arcsinh} iz}{i}$$

recommended by Kahan [25].

The branch cut for the arc sine function is in two pieces: one along the negative real axis to the left of -1 (inclusive), continuous with quadrant II, and one along the positive real axis to the right of 1 (inclusive), continuous with quadrant IV. The range is that strip of the complex plane containing numbers whose real part is between $-\pi/2$ and $\pi/2$. A number with real part equal to $-\pi/2$ is in the range if and only if its imaginary part is non-negative; a number with real part equal to $\pi/2$ is in the range if and only if its imaginary part is non-positive.

`acos` The following definition for arc cosine determines the range and branch cuts:

$$\arccos z = -i \log\left(z + i\sqrt{1 - z^2}\right)$$

or, which is equivalent,

$$\arccos z = \tfrac{\pi}{2} - \arcsin z$$

The branch cut for the arc cosine function is in two pieces: one along the negative real axis to the left of -1 (inclusive), continuous with quadrant II, and one along the positive real axis to the right of 1 (inclusive), continuous with quadrant IV. This is the same branch cut as for arc sine. The range is that strip of the complex plane containing numbers whose real part is between zero and π. A number with real part equal to zero is in the range if and only if its imaginary part is non-negative; a number with real part equal to π is in the range if and only if its imaginary part is non-positive.

`atan` The following definition for (one-argument) arc tangent determines the range and branch cuts:

X3J13 voted in January 1989 to replace the formula shown above with the formula

$$\arctan z = \frac{\log(1 + iz) - \log(1 - iz)}{2i}$$

This is equivalent to the formula

$$\arctan z = \frac{\operatorname{arctanh} iz}{i}$$

recommended by Kahan [25]. It causes the upper branch cut to be continuous with quadrant I rather than quadrant II, and the lower branch cut to be continuous with quadrant III rather than quadrant IV; otherwise it agrees with the formula of the first edition. Therefore this change alters the result returned by `atan` only for arguments on the positive imaginary axis that are of magnitude greater than 1. The full description for this new formula is as follows.

The branch cut for the arc tangent function is in two pieces: one along the positive imaginary axis above i (exclusive), continuous with quadrant I, and one along the negative imaginary axis below $-i$ (exclusive), continuous with quadrant III. The points i and $-i$ are excluded from the domain. The range is that strip of the complex plane containing numbers whose real part is between $-\pi/2$ and $\pi/2$. A number with real part equal to $-\pi/2$ is in the range if and only if its imaginary part is strictly negative; a number with real part equal to $\pi/2$ is in the range if and only if its imaginary part is strictly positive. Thus the range of the arc tangent function is *not* identical to that of the arc sine function.

`asinh` The following definition for the inverse hyperbolic sine determines the range and branch cuts:

$$\operatorname{arcsinh} z = \log\left(z + \sqrt{1 + z^2}\right)$$

The branch cut for the inverse hyperbolic sine function is in two pieces: one along the positive imaginary axis above i (inclusive), continuous with quadrant I, and one along the negative imaginary axis below $-i$

(inclusive), continuous with quadrant III. The range is that strip of the complex plane containing numbers whose imaginary part is between $-\pi/2$ and $\pi/2$. A number with imaginary part equal to $-\pi/2$ is in the range if and only if its real part is non-positive; a number with imaginary part equal to $\pi/2$ is in the range if and only if its real part is non-negative.

`acosh` The following definition for the inverse hyperbolic cosine determines the range and branch cuts:

$$\operatorname{arccosh} z = \log\left(z + (z+1)\sqrt{(z-1)/(z+1)}\right)$$

Kahan [25] suggests the formula

$$\operatorname{arccosh} z = 2\log\left(\sqrt{(z+1)/2} + \sqrt{(z-1)/2}\right)$$

pointing out that it yields the same principal value but eliminates a gratuitous removable singularity at $z = -1$. A proposal was submitted to X3J13 in September 1989 to replace the formula `acosh` with that recommended by Kahan. There is a good possibility that it will be adopted.

The branch cut for the inverse hyperbolic cosine function lies along the real axis to the left of 1 (inclusive), extending indefinitely along the negative real axis, continuous with quadrant II and (between 0 and 1) with quadrant I. The range is that half-strip of the complex plane containing numbers whose real part is non-negative and whose imaginary part is between $-\pi$ (exclusive) and π (inclusive). A number with real part zero is in the range if its imaginary part is between zero (inclusive) and π (inclusive).

`atanh` The following definition for the inverse hyperbolic tangent determines the range and branch cuts:

WARNING! *The formula shown above for hyperbolic arc tangent is incorrect.* It is not a matter of incorrect branch cuts; it simply does not compute anything like a hyperbolic arc tangent. This unfortunate error in the first edition was the result of mistranscribing a (correct) APL formula from Penfield's paper [36]. The formula should have been transcribed as

$$\operatorname{arctanh} z = \log\left((1+z)\sqrt{1/(1-z^2)}\right)$$

A proposal was submitted to X3J13 in September 1989 to replace the formula `atanh` with that recommended by Kahan [25]:

$$\operatorname{arctanh} z = \frac{(\log(1+z) - \log(1-z))}{2}$$

There is a good possibility that it will be adopted. If it is, the complete description of the branch cuts of `atanh` will then be as follows.

The branch cut for the inverse hyperbolic tangent function is in two pieces: one along the negative real axis to the left of -1 (inclusive), continuous with quadrant II, and one along the positive real axis to the right of 1 (inclusive), continuous with quadrant IV. The points -1 and 1 are excluded from the domain. The range is that strip of the complex plane containing numbers whose imaginary part is between $-\pi/2$ and $\pi/2$. A number with imaginary part equal to $-\pi/2$ is in the range if and only if its real part is strictly positive; a number with imaginary part equal to $\pi/2$ is in the range if and only if its real part is strictly negative. Thus the range of the inverse hyperbolic tangent function is *not* the same as that of the inverse hyperbolic sine function.

With these definitions, the following useful identities are obeyed throughout the applicable portion of the complex domain, even on the branch cuts:

$\sin iz = i\sinh z$	$\sinh iz = i\sin z$	$\arctan iz = i\operatorname{arctanh} z$
$\cos iz = \cosh z$	$\cosh iz = \cos z$	$\operatorname{arcsinh} iz = i\arcsin z$
$\tan iz = i\tanh z$	$\arcsin iz = i\operatorname{arcsinh} z$	$\operatorname{arctanh} iz = i\arctan z$

I thought it would be useful to provide some graphs illustrating the behavior of the irrational and transcendental functions in the complex plane. It also provides an opportunity to show off the Common Lisp code that was used to generate them.

Imagine the complex plane to be decorated as follows. The real and imaginary axes are painted with thick lines. Parallels from the axes on both sides at distances of 1, 2, and 3 are painted with thin lines; these parallels are doubly infinite lines, as are the axes. Four annuli (rings) are painted in gradated shades of gray. Ring 1, the inner ring, consists of points whose

radial distances from the origin lie in the range $[1/4, 1/2]$; ring 2 is in the radial range $[3/4, 1]$; ring 3, in the range $[\pi/2, 2]$; and ring 4, in the range $[3, \pi]$. Ring j is divided into 2^{j+1} equal sectors, with each sector painted a different shade of gray, darkening as one proceeds counterclockwise from the positive real axis.

We can illustrate the behavior of a numerical function f by considering how it maps the complex plane to itself. More specifically, consider each point z of the decorated plane. We decorate a new plane by coloring the point $f(z)$ with the same color that point z had in the original decorated plane. In other words, the newly decorated plane illustrates how the f maps the axes, other horizontal and vertical lines, and annuli.

In each figure we will show only a fragment of the complex plane, with the real axis horizontal in the usual manner ($-\infty$ to the left, $+\infty$ to the right) and the imaginary axis vertical ($-\infty i$ below, $+\infty i$ above). Each fragment shows a region containing points whose real and imaginary parts are in the range $[-4.1, 4.1]$. The axes of the new plane are shown as very thin lines, with large tick marks at integer coordinates and somewhat smaller tick marks at multiples of $\pi/2$.

Figure 12.1 shows the result of plotting the `identity` function (quite literally); the graph exhibits the decoration of the original plane.

Figures 12.2 through 12.20 show the graphs for the functions `sqrt`, `exp`, `log`, `sin`, `asin`, `cos`, `acos`, `tan`, `atan`, `sinh`, `asinh`, `cosh`, `acosh`, `tanh`, and `atanh`, and as a bonus, the graphs for the functions $\sqrt{1-z^2}$, $\sqrt{1+z^2}$, $(z-1)/(z+1)$, and $(1+z)/(1-z)$. All of these are related to the trigonometric functions in various ways. For example, if $f(z) = (z-1)/(z+1)$, then $\tanh z = f(e^{2z})$, and if $g(z) = \sqrt{1-z^2}$, then $\cos z = g(\sin z)$. It is instructive to examine the graph for $\sqrt{1-z^2}$ and try to visualize how it transforms the graph for `sin` into the graph for `cos`.

Each figure is accompanied by a commentary on what maps to what and other interesting features. None of this material is terribly new; much of it may be found in any good textbook on complex analysis. I believe that the particular form in which the graphs are presented is novel, as well as the fact that the graphs have been generated as PostScript [1] code by Common Lisp code. This PostScript code was then fed directly to the typesetting equipment that set the pages for this book. Samples of this PostScript code follow the figures themselves, after which the code for the entire program is presented.

In the commentaries that accompany the figures I sometimes speak of

mapping the points $\pm\infty$ or $\pm\infty i$. When I say that function f maps $+\infty$ to a certain point z, I mean that

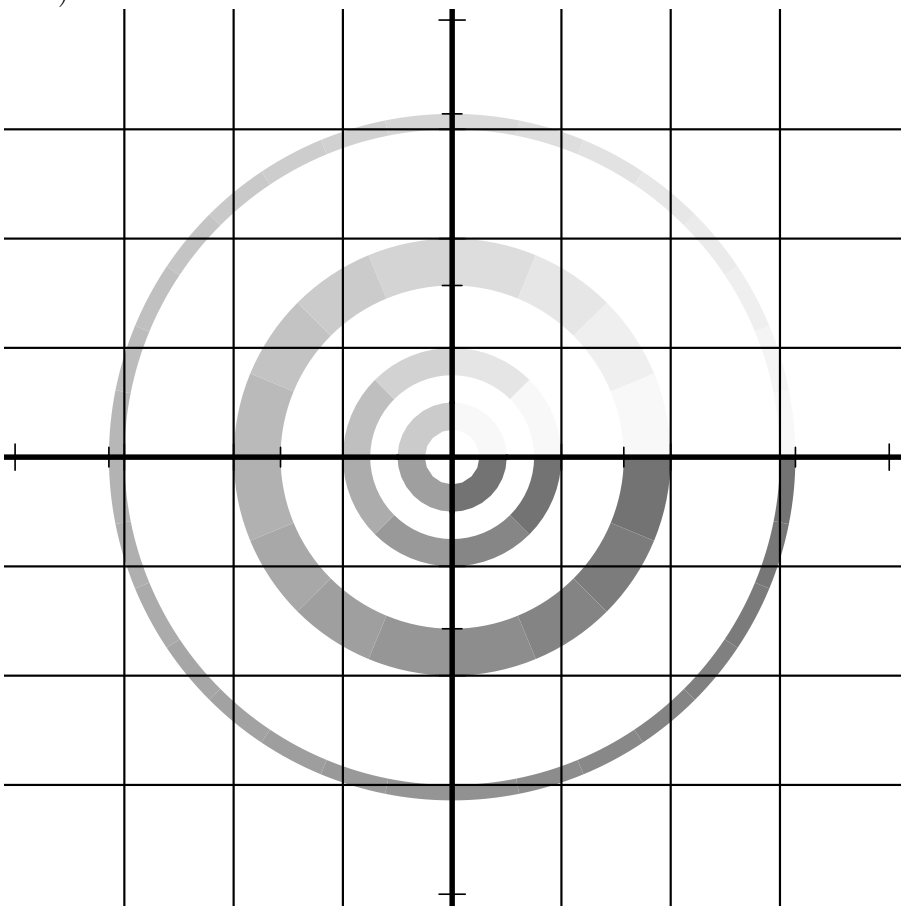
$z = \lim_{x\to+\infty} f(x + 0\,i)$

Similarly, when I say that f maps $-\infty i$ to z, I mean that

$z = \lim_{y\to-\infty} f(0 + yi)$

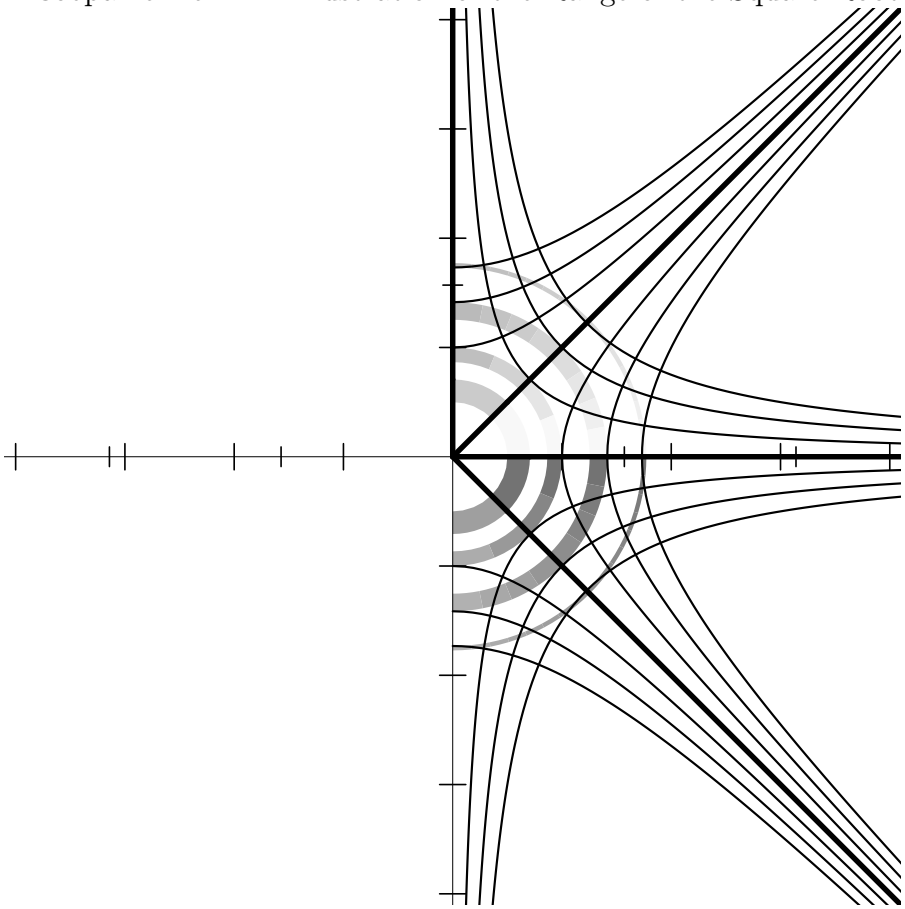
In other words, I am considering a limit as one travels out along one of the main axes. I also speak in a similar manner of mapping *to* one of these infinities.

Изображение 12.1: Initial Decoration of the Complex Plane (Identity Function)



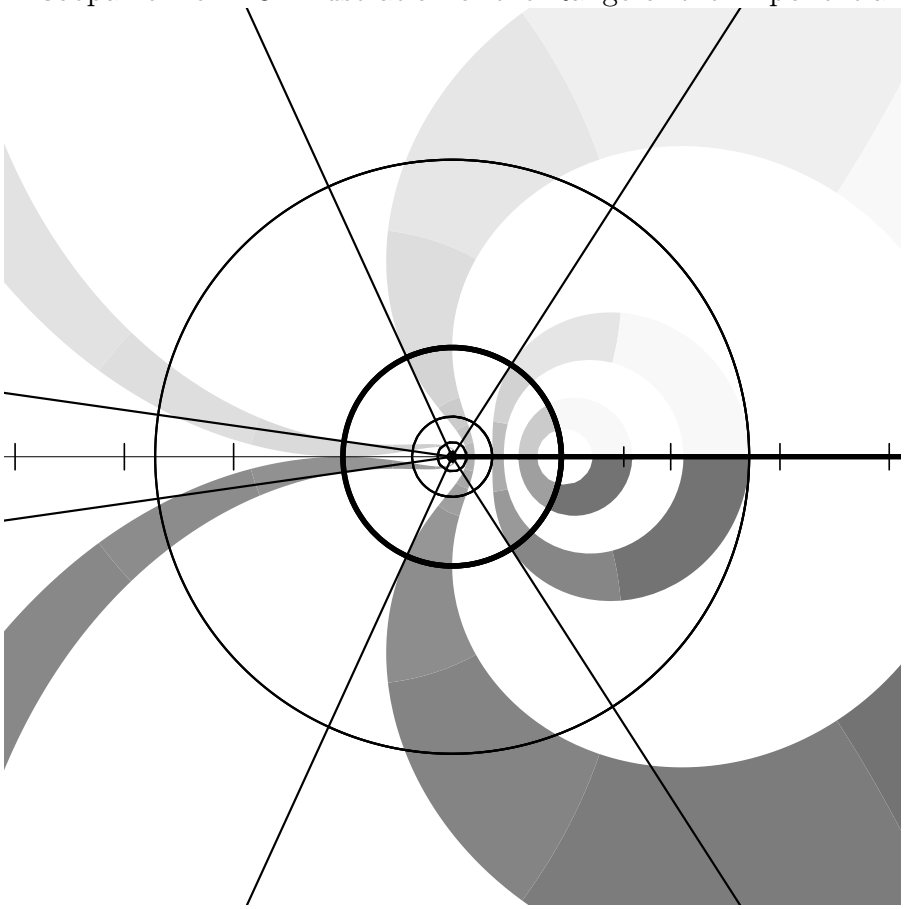
This figure was produced in exactly the same manner as succeeding figures, simply by plotting the function `identity` instead of a numerical function. Thus the first of these figures was produced by the last function of the first edition. I knew it would come in handy someday!

Изображение 12.2: Illustration of the Range of the Square Root Function



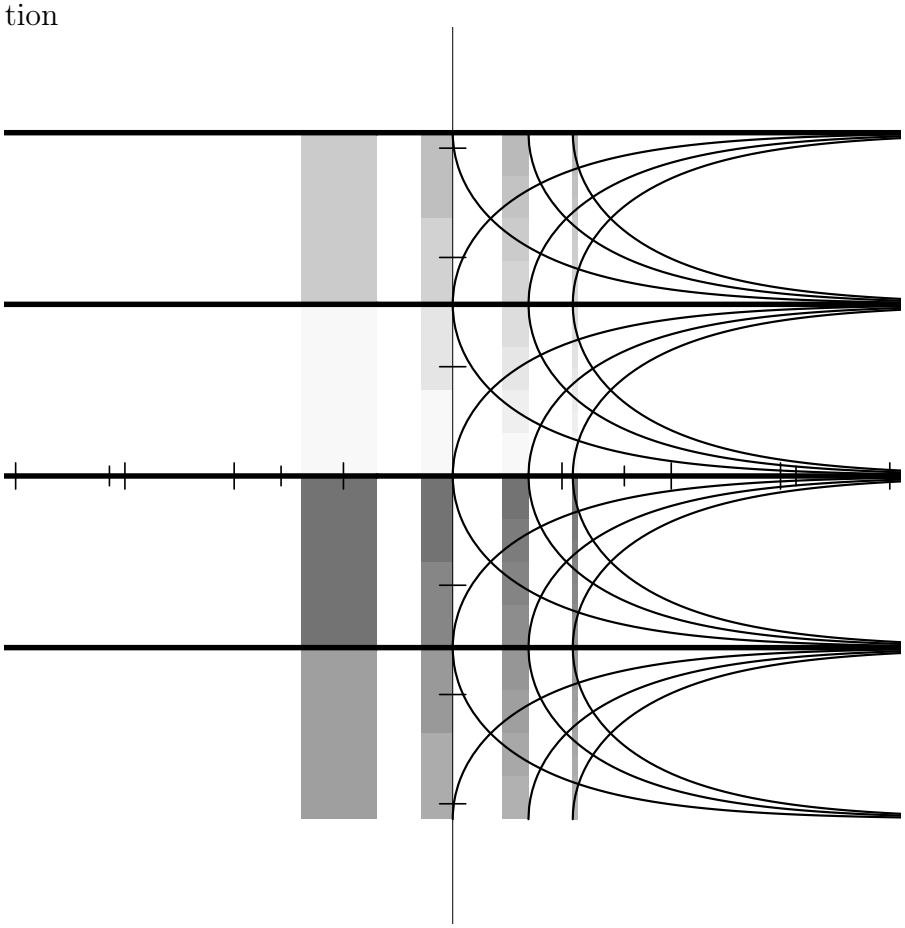
The `sqrt` function maps the complex plane into the right half of the plane by slitting it along the negative real axis and then sweeping it around as if half-closing a folding fan. The fan also shrinks, as if it were made of cotton and had gotten wetter at the periphery than at the center. The positive real axis is mapped onto itself. The negative real axis is mapped onto the positive imaginary axis (but if minus zero is supported, then $-x + 0i$ is mapped onto the positive imaginary axis and $-x - 0i$ onto the negative imaginary axis, assuming $x > 0$). The positive imaginary axis is mapped onto the northeast diagonal, and the negative imaginary axis onto the southeast diagonal. More generally, lines are mapped to rectangular hyperbolas (or fragments thereof) centered on the origin; lines through the origin are mapped to degenerate hyperbolas (perpendicular lines through the origin).

Изображение 12.3: Illustration of the Range of the Exponential Function



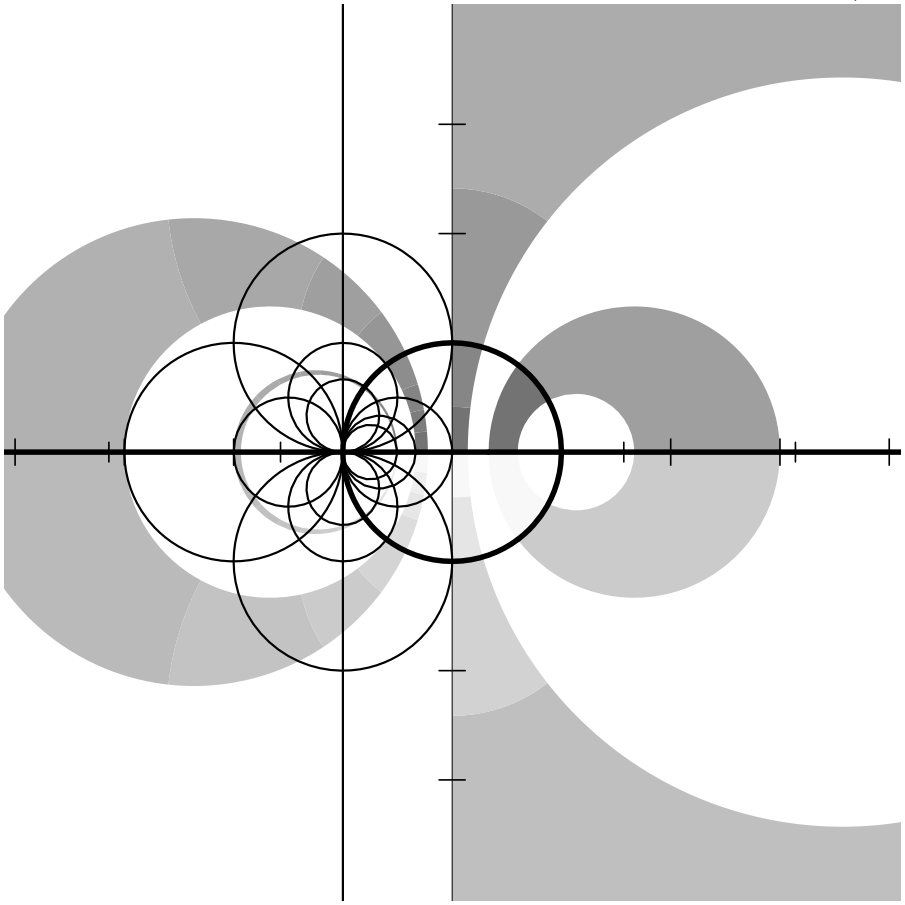
The `exp` function maps horizontal lines to radii and maps vertical lines to circles centered at the origin. The origin is mapped to 1. (It is instructive to compare this graph with those of other functions that map the origin to 1, for example $(1+z)/(1-z)$, $\cos z$, and $\sqrt{1-z^2}$.) The entire real axis is mapped to the positive real axis, with $-\infty$ mapping to the origin and $+\infty$ to itself. The imaginary axis is mapped to the unit circle with infinite multiplicity (period 2π); therefore the mapping of the imaginary infinities $\pm\infty i$ is indeterminate. It follows that the entire left half-plane is mapped to the interior of the unit circle, and the right half-plane is mapped to the exterior of the unit circle. A line at any angle other than horizontal or vertical is mapped to a logarithmic spiral (but this is not illustrated here).

Изображение 12.4: Illustration of the Range of the Natural Logarithm Function



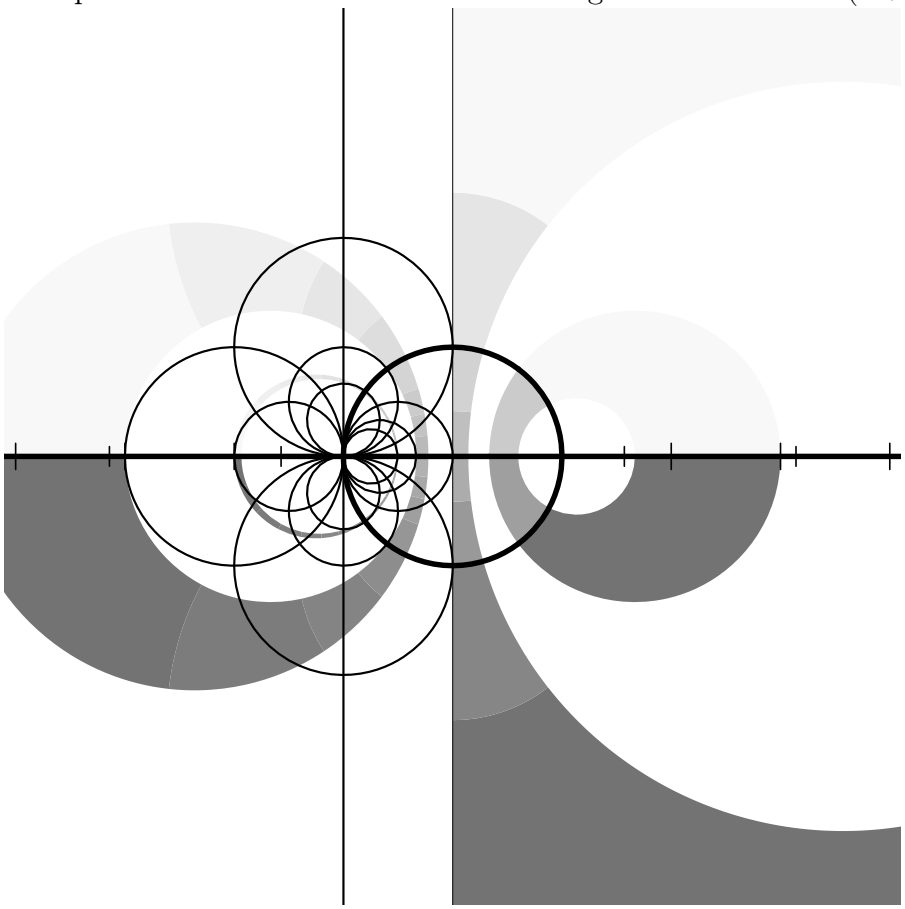
The `log` function, which is the inverse of `exp`, naturally maps radial lines to horizontal lines and circles centered at the origin to vertical lines. The interior of the unit circle is thus mapped to the entire left half-plane, and the exterior of the unit circle is mapped to the right half-plane. The positive real axis is mapped to the entire real axis, and the negative real axis to a horizontal line of height π. The positive and negative imaginary axes are mapped to horizontal lines of height $\pm\pi/2$. The origin is mapped to $-\infty$.

Изображение 12.5: Illustration of the Range of the Function $(z-1)/(z+1)$



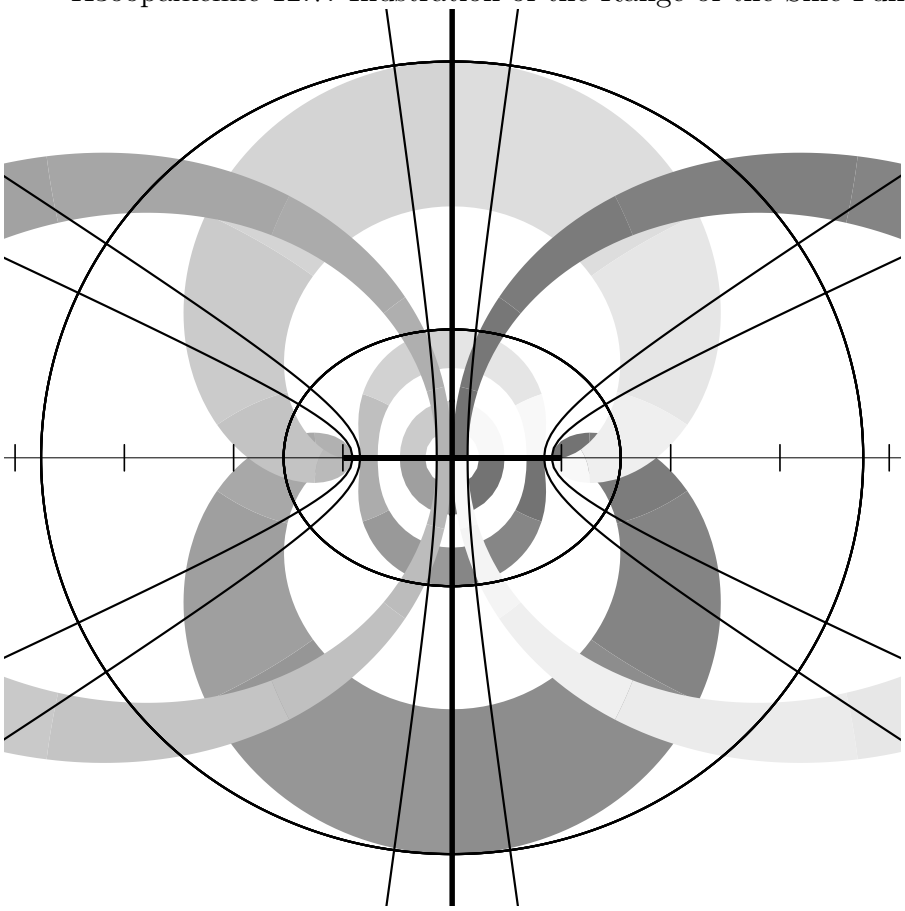
A line is a degenerate circle with infinite radius; when I say "circles" here I also mean lines. Then $(z-1)/(z+1)$ maps circles into circles. All circles through -1 become lines; all lines become circles through 1. The real axis is mapped onto itself: 1 to the origin, the origin to -1, -1 to infinity, and infinity to 1. The imaginary axis becomes the unit circle; i is mapped to itself, as is $-i$. Thus the entire right half-plane is mapped to the interior of the unit circle, the unit circle interior to the left half-plane, the left half-plane to the unit circle exterior, and the unit circle exterior to the right half-plane. Imagine the complex plane to be a vast sea. The Colossus of Rhodes straddles the origin, its left foot on i and its right foot on $-i$. It bends down and briefly paddles water between its legs so furiously that the water directly beneath is pushed out into the entire area behind it; much that was behind swirls forward to either side; and all that was before is sucked in to lie between its feet.

Изображение 12.6: Illustration of the Range of the Function $(1+z)/(1-z)$



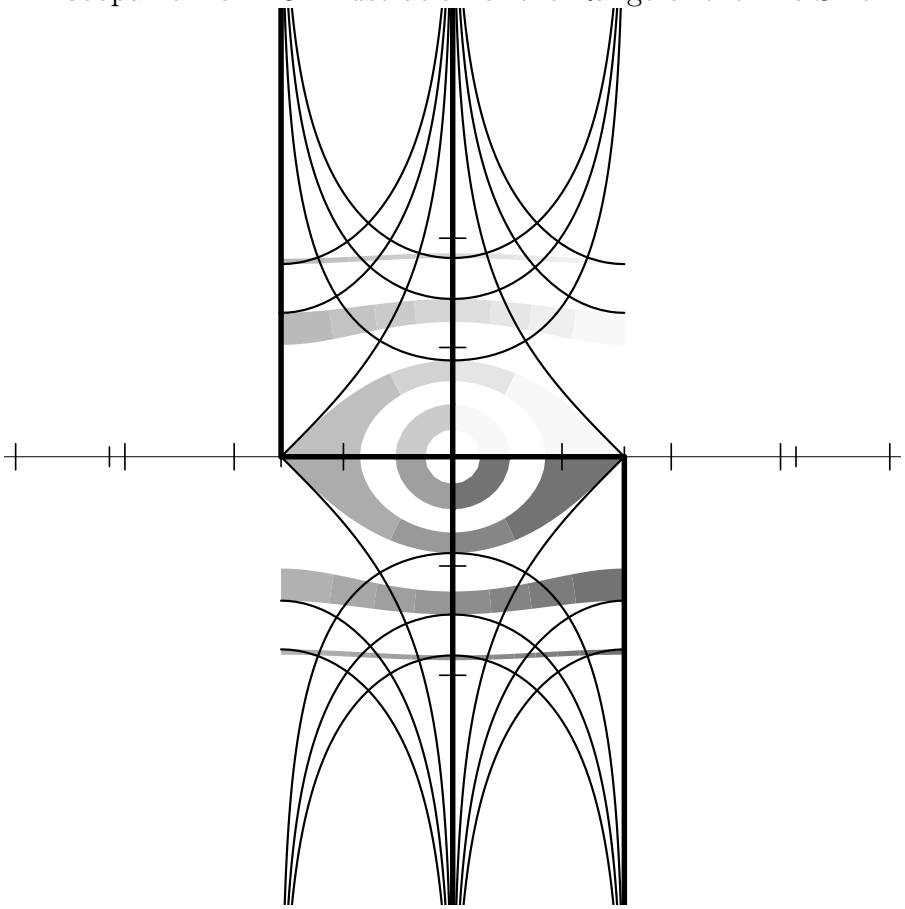
The function $h(z) = (1+z)/(1-z)$ is the inverse of $f(z) = (z-1)/(z+1)$; that is, $h(f(z)) = f(h(z)) = z$. At first glance, the graph of h appears to be that of f flipped left-to-right, or perhaps reflected in the origin, but careful consideration of the shaded annuli reveals that this is not so; something more subtle is going on. Note that $f(f(z)) = h(h(z)) = g(z) = -1/z$. The functions f, g, h, and the identity function thus form a group under composition, isomorphic to the group of the cyclic permutations of the points -1, 0, 1, and ∞, as indeed these functions accomplish the four possible cyclic permutations on those points. This function group is a subset of the group of bilinear transformations $(az+b)/(cz+d)$, all of which are conformal (angle-preserving) and map circles onto circles. Now, doesn't that tangle of circles through -1 look like something the cat got into?

Изображение 12.7: Illustration of the Range of the Sine Function



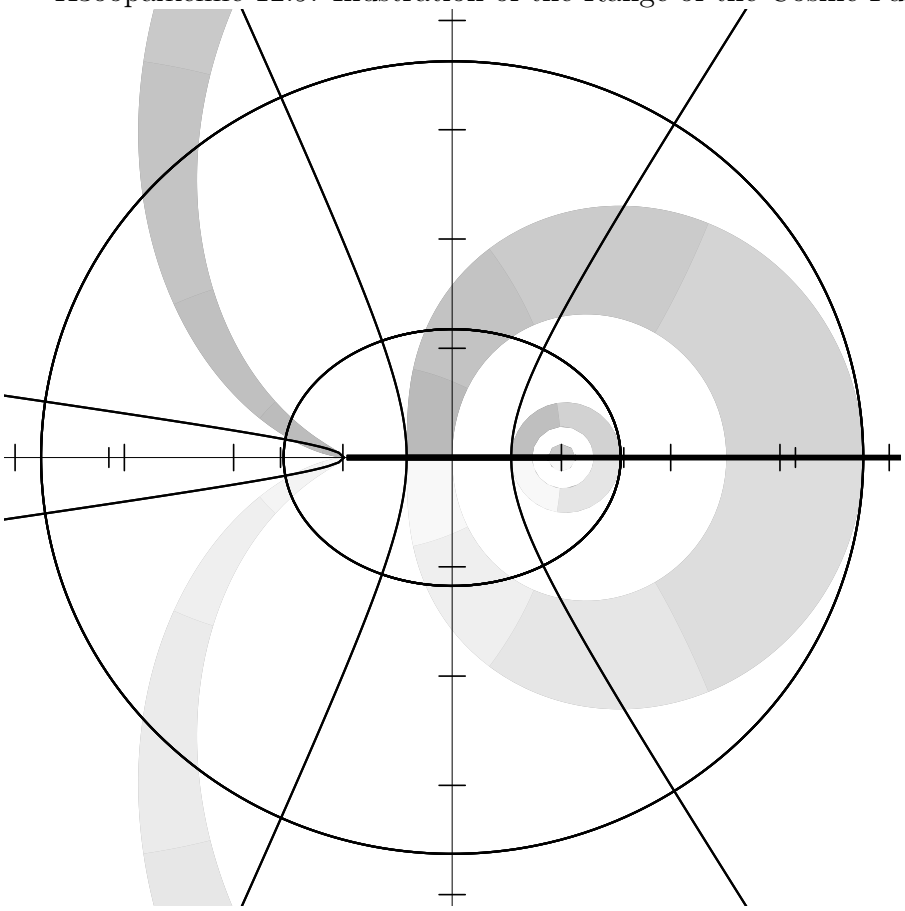
We are used to seeing `sin` looking like a wiggly ocean wave, graphed vertically as a function of the real axis only. Here is a different view. The entire real axis is mapped to the segment $[-1, 1]$ of the real axis with infinite multiplicity (period 2π). The imaginary axis is mapped to itself as if by `sinh` considered as a real function. The origin is mapped to itself. Horizontal lines are mapped to ellipses with foci at ± 1 (note that two horizontal lines equidistant from the real axis will map onto the same ellipse). Vertical lines are mapped to hyperbolas with the same foci. There is a curious accident: the ellipse for horizontal lines at distance ± 1 from the real axis appears to intercept the real axis at $\pm\pi/2 \approx \pm 1.57\ldots$ but this is not so; the intercepts are actually at $\pm(e + 1/e)/2 \approx \pm 1.54\ldots$.

Изображение 12.8: Illustration of the Range of the Arc Sine Function



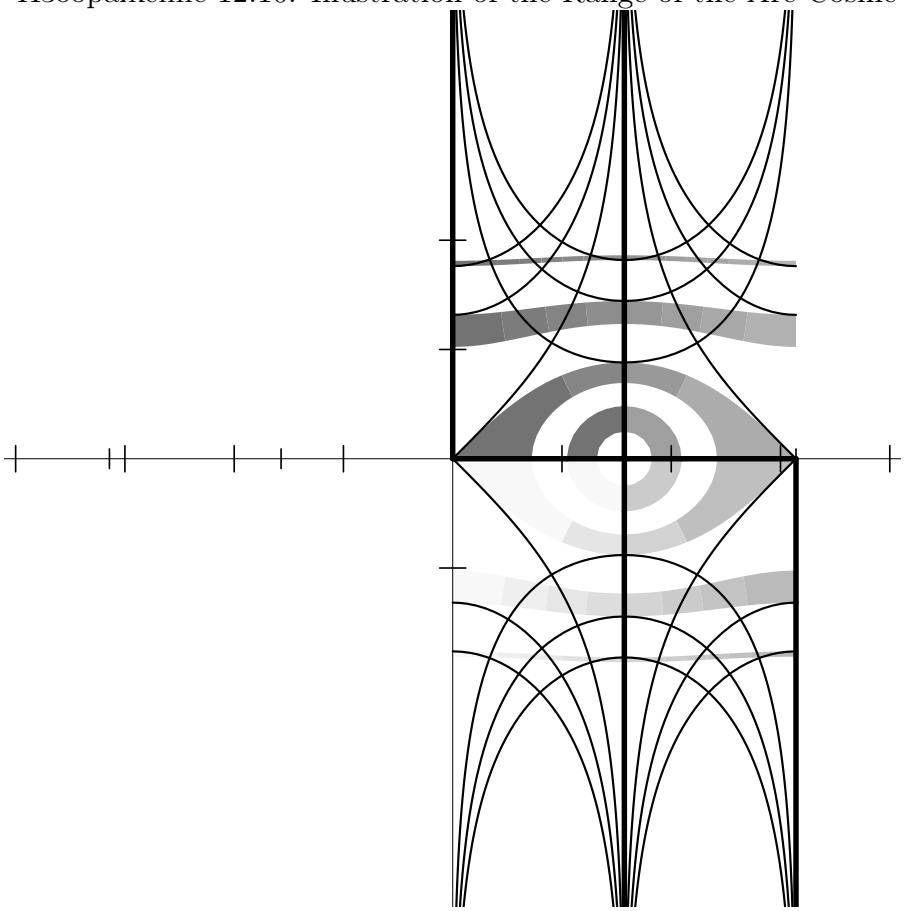
Just as `sin` grabs horizontal lines and bends them into elliptical loops around the origin, so its inverse `asin` takes annuli and yanks them more or less horizontally straight. Because sine is not injective, its inverse as a function cannot be surjective. This is just a highfalutin way of saying that the range of the `asin` function doesn't cover the entire plane but only a strip π wide; arc sine as a one-to-many relation would cover the plane with an infinite number of copies of this strip side by side, looking for all the world like the tail of a peacock with an infinite number of feathers. The imaginary axis is mapped to itself as if by `asinh` considered as a real function. The real axis is mapped to a bent path, turning corners at $\pm\pi/2$ (the points to which ± 1 are mapped); $+\infty$ is mapped to $\pi/2 - \infty i$, and $-\infty$ to $-\pi/2 + \infty i$.

Изображение 12.9: Illustration of the Range of the Cosine Function



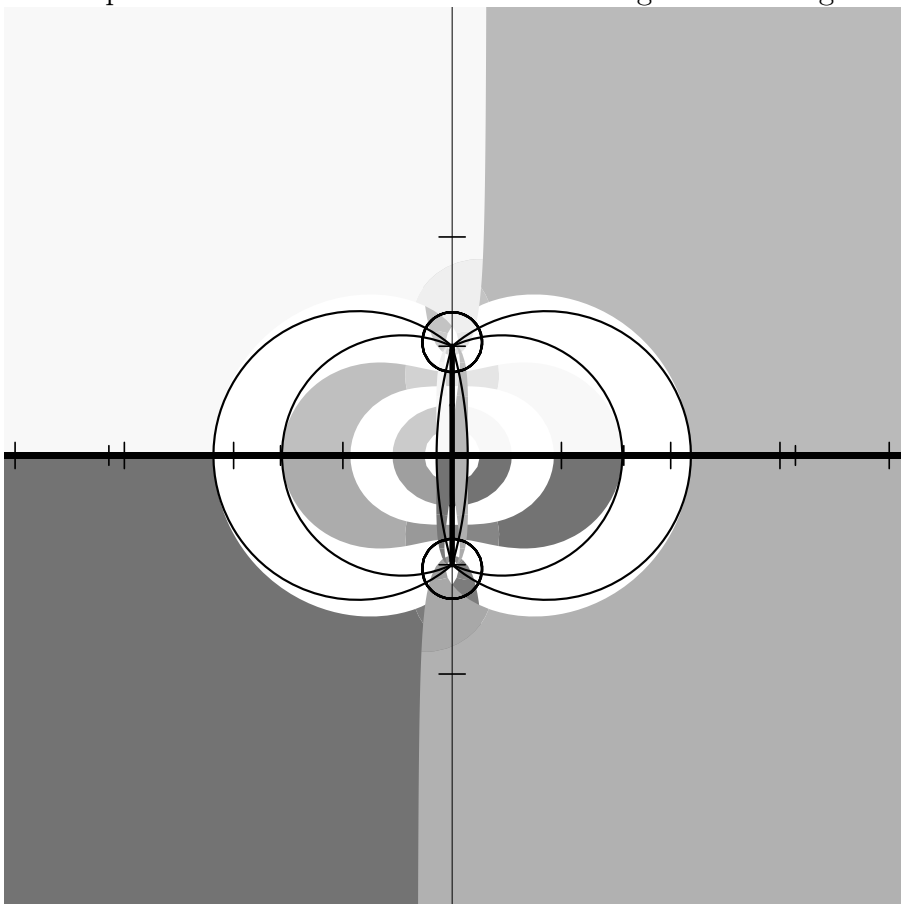
We are used to seeing `cos` looking exactly like `sin`, a wiggly ocean wave, only displaced. Indeed the complex mapping of `cos` is also similar to that of `sin`, with horizontal and vertical lines mapping to the same ellipses and hyperbolas with foci at ± 1, although mapping to them in a different manner, to be sure. The entire real axis is again mapped to the segment $[-1, 1]$ of the real axis, but each half of the imaginary axis is mapped to the real axis to the right of 1 (as if by `cosh` considered as a real function). Therefore $\pm\infty i$ both map to $+\infty$. The origin is mapped to 1. Whereas `sin` is an odd function, `cos` is an even function; as a result *two* points in each annulus, one the negative of the other, are mapped to the same shaded point in this graph; the shading shown here is taken from points in the original upper half-plane.

Изображение 12.10: Illustration of the Range of the Arc Cosine Function



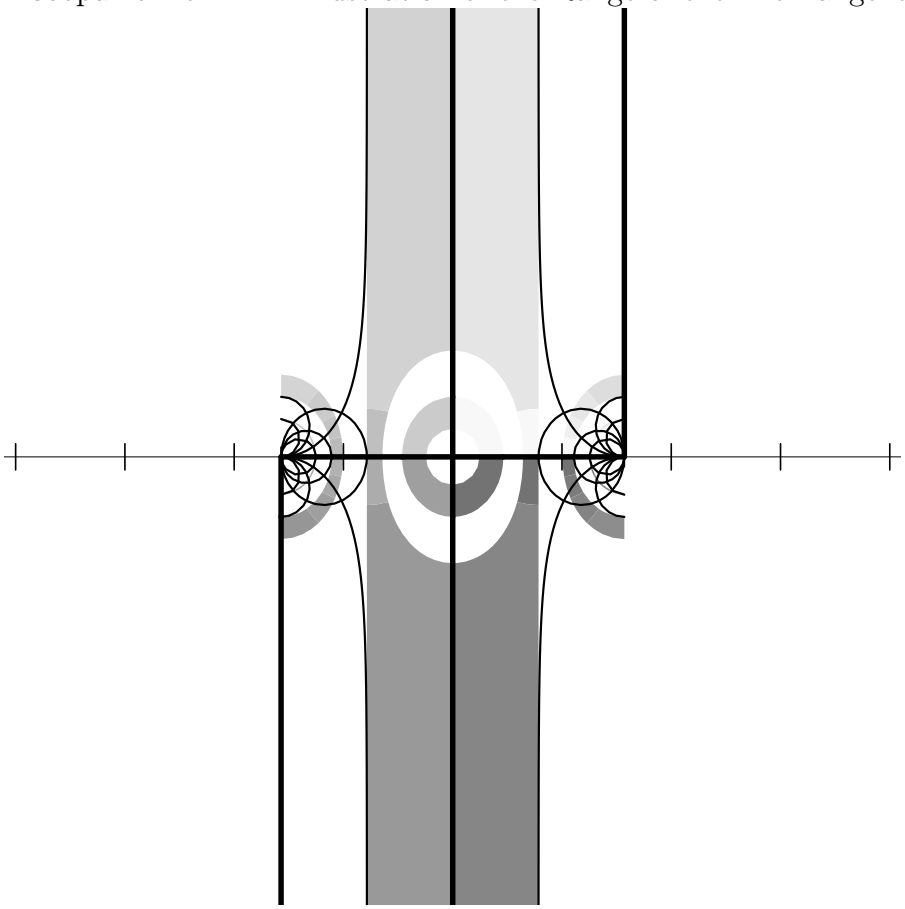
The graph of `acos` is very much like that of `asin`. One might think that our nervous peacock has shuffled half a step to the right, but the shading on the annuli shows that we have instead caught the bird exactly in mid-flight while doing a cartwheel. This is easily understood if we recall that $\arccos z = (\pi/2) - \arcsin z$; negating $\arcsin z$ rotates it upside down, and adding the result to $\pi/2$ translates it $\pi/2$ to the right. The imaginary axis is mapped upside down to the vertical line at $\pi/2$. The point $+1$ is mapped to the origin, and -1 to π. The image of the real axis is again cranky; $+\infty$ is mapped to $+\infty i$, and $-\infty$ to $\pi - \infty i$.

Изображение 12.11: Illustration of the Range of the Tangent Function



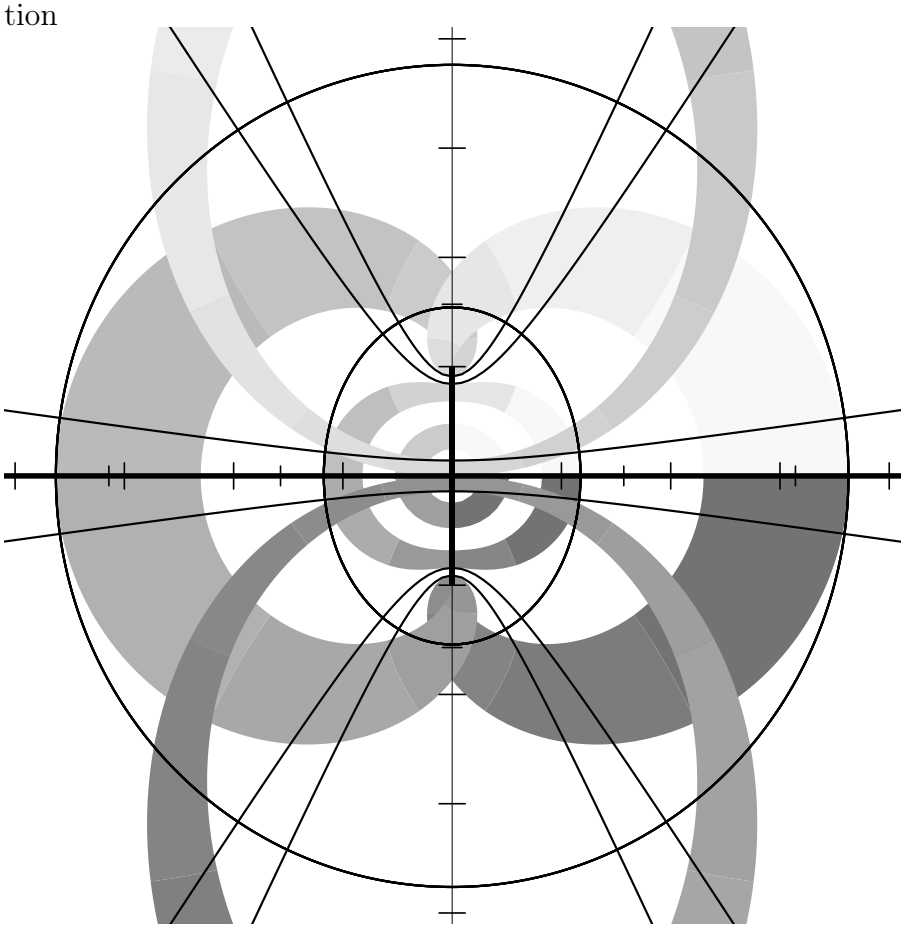
The usual graph of `tan` as a real function looks like an infinite chorus line of disco dancers, left hands pointed skyward and right hands to the floor. The `tan` function is the quotient of `sin` and `cos` but it doesn't much look like either except for having period 2π. This goes for the complex plane as well, although the swoopy loops produced from the annulus between $\pi/2$ and 2 look vaguely like those from the graph of `sin` inside out. The real axis is mapped onto itself with infinite multiplicity (period 2π). The imaginary axis is mapped backwards onto $[-i, i]$: $+\infty i$ is mapped to $-i$ and $-\infty i$ to $+i$. Horizontal lines below or above the real axis become circles surrounding $+i$ or $-i$, respectively. Vertical lines become circular arcs from $+i$ to $-i$; two vertical lines separated by $(2k+1)\pi$ for integer k together become a complete circle. It seems that two arcs shown hit the real axis at $\pm\pi/2 = \pm 1.57\ldots$ but that is a coincidence; they really hit the axis at $\pm\tan 1 = 1.55\ldots$.

Изображение 12.12: Illustration of the Range of the Arc Tangent Function



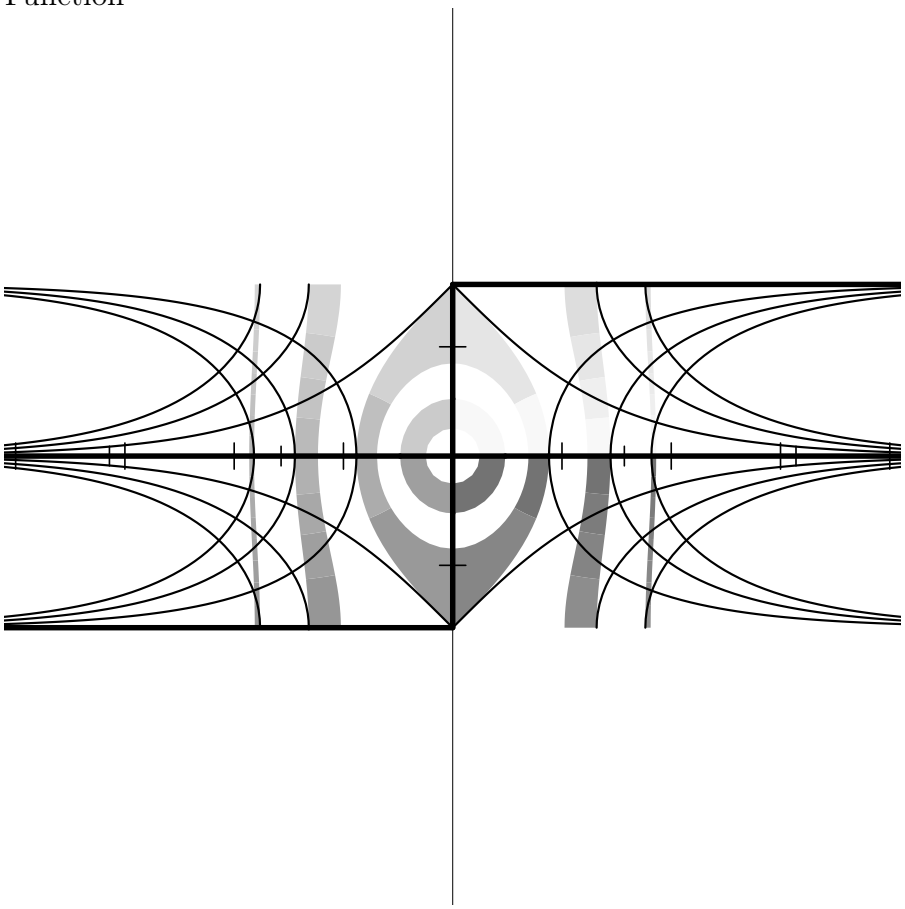
All I can say is that this peacock is a horse of another color. At first glance, the axes seem to map in the same way as for `asin` and `acos`, but look again: this time it's the imaginary axis doing weird things. All infinities map multiply to the points $(2k+1)\pi/2$; within the strip of principal values we may say that the real axis is mapped to the interval $[-\pi/2, +\pi/2]$ and therefore $-\infty$ is mapped to $-\pi/2$ and $+\infty$ to $+\pi/2$. The point $+i$ is mapped to $+\infty i$, and $-i$ to $-\infty i$, and so the imaginary axis is mapped into three pieces: the segment $[-\infty i, -i]$ is mapped to $[\pi/2, \pi/2 - \infty i]$; the segment $[-i, i]$ is mapped to the imaginary axis $[-\infty i, +\infty i]$; and the segment $[+i, +\infty i]$ is mapped to $[-\pi/2 + \infty i, -\pi/2]$.

Изображение 12.13: Illustration of the Range of the Hyperbolic Sine Function



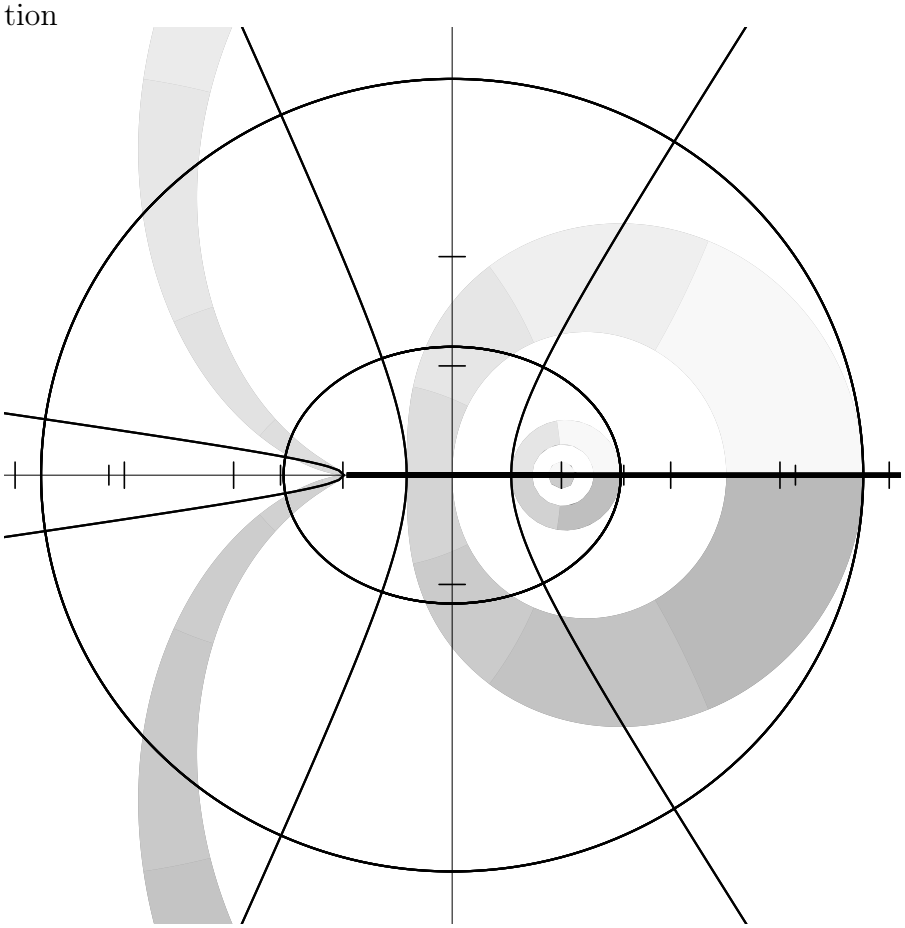
It would seem that the graph of `sinh` is merely that of `sin` rotated 90 degrees. If that were so, then we would have $\sinh z = i\sin z$. Careful inspection of the shading, however, reveals that this is not quite the case; in both graphs the lightest and darkest shades, which initially are adjacent to the positive real axis, remain adjacent to the positive real axis in both cases. To derive the graph of `sinh` from `sin` we must therefore first rotate the complex plane by -90 degrees, then apply `sin`, then rotate the result by 90 degrees. In other words, $\sinh z = i\sin(-i)z$; consistently replacing z with iz in this formula yields the familiar identity $\sinh iz = i\sin z$.

Изображение 12.14: Illustration of the Range of the Hyperbolic Arc Sine Function



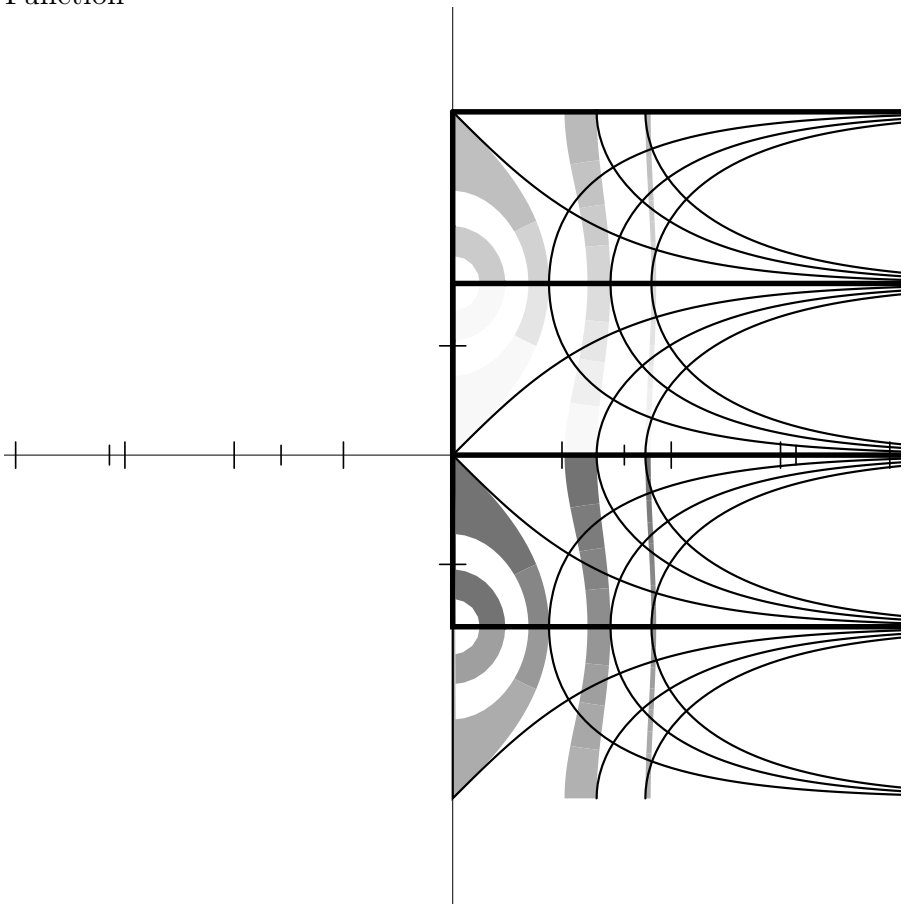
The peacock sleeps. Because $\operatorname{arcsinh} iz = i \arcsin z$, the graph of `asinh` is related to that of `asin` by pre- and post-rotations of the complex plane in the same way as for `sinh` and `sin`.

Изображение 12.15: Illustration of the Range of the Hyperbolic Cosine Function



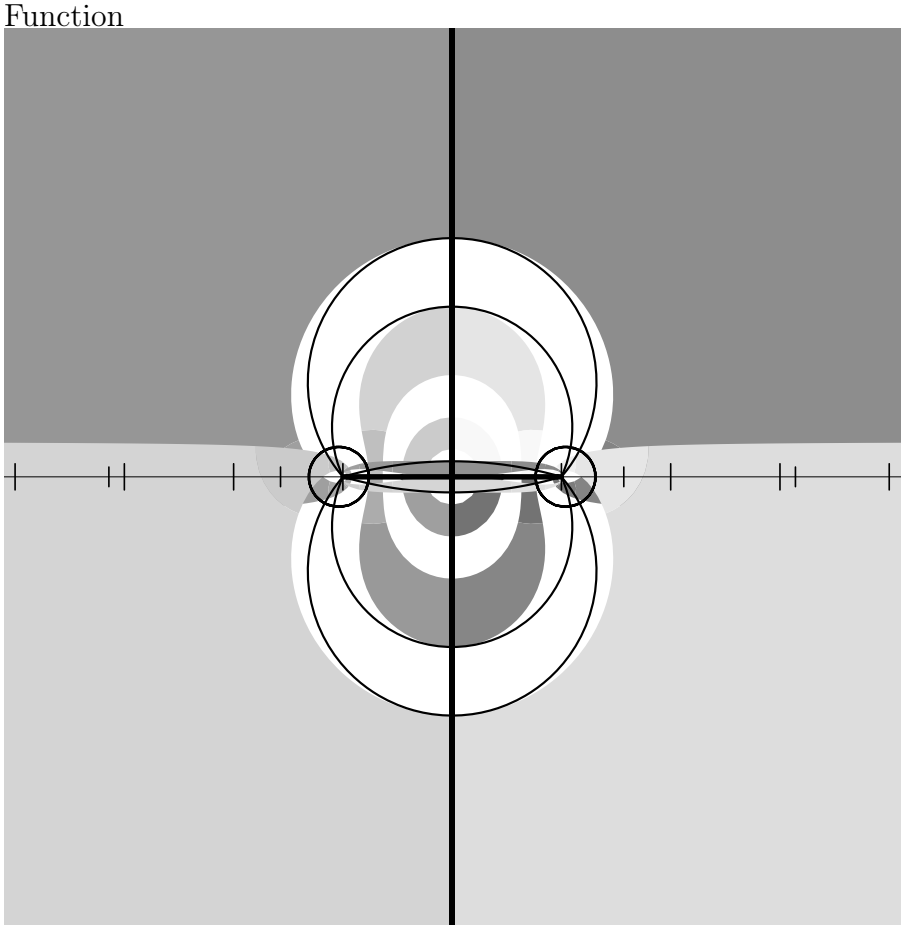
The graph of `cosh` does *not* look like that of `cos` rotated 90 degrees; instead it looks like that of `cos` unrotated. That is because $\cosh iz$ is not equal to $i\cos z$; rather, $\cosh iz = \cos z$. Interpreted, that means that the shading is pre-rotated but there is no post-rotation.

Изображение 12.16: Illustration of the Range of the Hyperbolic Arc Cosine Function



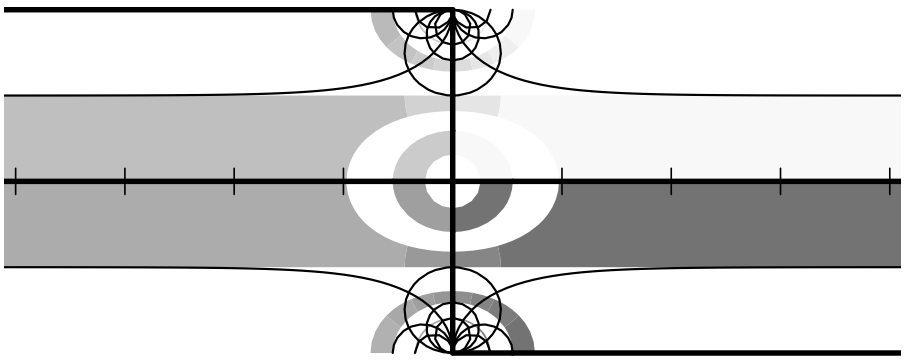
Hmm—I'd rather not say what happened to this peacock. This feather looks a bit mangled. Actually it is all right—the principal value for `acosh` is so chosen that its graph does not look simply like a rotated version of the graph of `acos`, but if all values were shown, the two graphs would fill the plane in repeating patterns related by a rotation.

Изображение 12.17: Illustration of the Range of the Hyperbolic Tangent Function



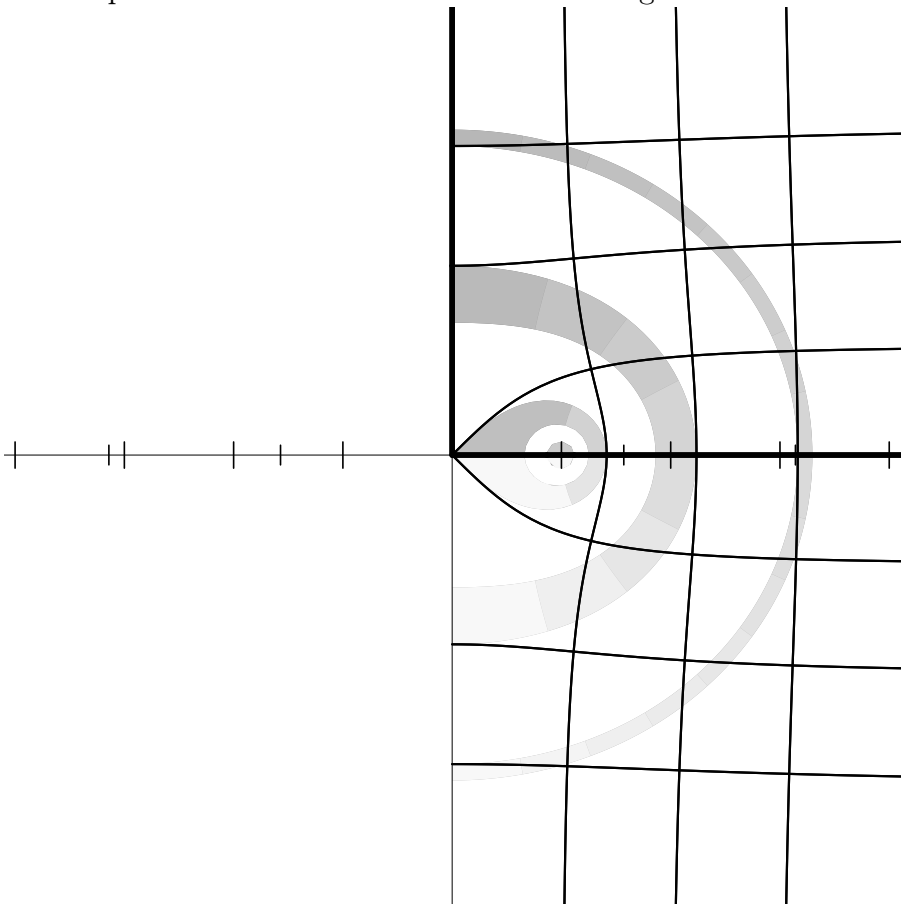
The diagram for `tanh` is simply that of `tan` turned on its ear: $i\tan z = \tanh iz$. The imaginary axis is mapped onto itself with infinite multiplicity (period 2π), and the real axis is mapped onto the segment $[-1,+1]$: $+\infty$ is mapped to $+1$, and $-\infty$ to -1. Vertical lines to the left or right of the real axis are mapped to circles surrounding -1 or 1, respectively. Horizontal lines are mapped to circular arcs anchored at -1 and $+1$; two horizontal lines separated by a distance $(2k+1)\pi$ for integer k are together mapped into a complete circle. How do we know these really are circles? Well, $\tanh z = ((\exp 2z)-1)/((\exp 2z)+1)$, which is the composition of the bilinear transform $(z-1)/(z+1)$, the exponential $\exp z$, and the magnification $2z$. Magnification maps lines to lines of the same slope; the exponential maps horizontal lines to circles and vertical lines to radial lines; and a bilinear transform maps generalized circles (including lines) to generalized circles. Q.E.D.

Изображение 12.18: Illustration of the Range of the Hyperbolic Arc Tangent Function



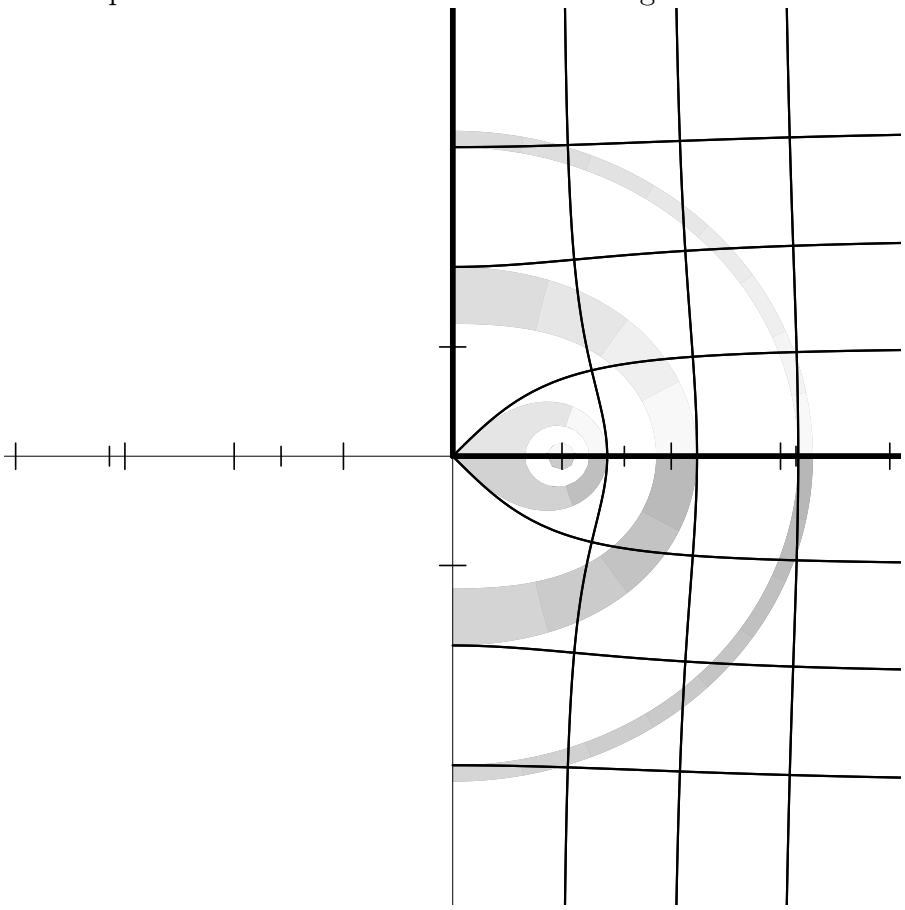
A sleeping peacock of another color: $\operatorname{arctanh} iz = i \arctan z$.

Изображение 12.19: Illustration of the Range of the Function $\sqrt{1-z^2}$



Here is a curious graph indeed for so simple a function! The origin is mapped to 1. The real axis segment $[0,1]$ is mapped backwards (and non-linearly) into itself; the segment $[1,+\infty]$ is mapped non-linearly onto the positive imaginary axis. The negative real axis is mapped to the same points as the positive real axis. Both halves of the imaginary axis are mapped into $[1,+\infty]$ on the real axis. Horizontal lines become vaguely vertical, and vertical lines become vaguely horizontal. Circles centered at the origin are transformed into Cassinian (half-)ovals; the unit circle is mapped to a (half-)lemniscate of Bernoulli. The outermost annulus appears to have its *inner* edge at π on the real axis and its *outer* edge at 3 on the imaginary axis, but this is another accident; the intercept on the real axis, for example, is not really at $\pi \approx 3.14\ldots$ but at $\sqrt{1-(3i)^2} = \sqrt{10} \approx 3.16\ldots$.

Изображение 12.20: Illustration of the Range of the Function $\sqrt{1+z^2}$



The graph of $q(z) = \sqrt{1+z^2}$ looks like that of $p(z) = \sqrt{1-z^2}$ except for the shading. You might not expect p and q to be related in the same way that `cos` and `cosh` are, but after a little reflection (or perhaps I should say, after turning it around in one's mind) one can see that $q(iz) = p(z)$. This formula is indeed of exactly the same form as $\cosh iz = \cos z$. The function $\sqrt{1+z^2}$ maps both halves of the real axis into $[1, +\infty]$ on the real axis. The segments $[0, i]$ and $[0, -i]$ of the imaginary axis are each mapped backwards onto segment $[0, 1]$ of the real axis; $[i, +\infty i]$ and $[-, -\infty i]$ are each mapped onto the positive imaginary axis (but if minus zero is supported then opposite sides of the imaginary axis map to opposite halves of the imaginary axis—for example, $q(+0 + 2i) = \sqrt{5}i$ but $q(-0 + 2i) = -\sqrt{5}i$).

Here is a sample of the PostScript code that generated figure 12.1, showing the initial scaling, translation, and clipping parameters; the code for one sector of the innermost annulus; and the code for the negative imaginary axis. Comment lines indicate how path or boundary segments were generated separately and then spliced (in order to allow for the places that a singularity might lurk, in which case the generating code can "inch up" to the problematical argument value).

The size of the entire PostScript file for the `identity` function was about 68 kilobytes (2757 lines, including comments). The smallest files were the plots for `atan` and `atanh`, about 65 kilobytes apiece; the largest were the plots for `sin`, `cos`, `sinh`, and `cosh`, about 138 kilobytes apiece.

```
% PostScript file for plot of function IDENTITY
% Plot is to fit in a region 4.666666666666667 inches square
%  showing axes extending 4.1 units from the origin.

40.97560975609756 40.97560975609756 scale
4.1 4.1 translate
newpath
  -4.1 -4.1 moveto
  4.1 -4.1 lineto
  4.1 4.1 lineto
  -4.1 4.1 lineto
  closepath
clip
% Moby grid for function IDENTITY
% Annulus 0.25 0.5 4 0.97 0.45
% Sector from 4.7124 to 6.2832 (quadrant 3)
newpath
  0.0 -0.25 moveto
  0.0 -0.375 lineto
  %middle radial
  0.0 -0.375 lineto
  0.0 -0.5 lineto
  %end radial
  0.0 -0.5 lineto
  0.092 -0.4915 lineto
  0.1843 -0.4648 lineto
  0.273 -0.4189 lineto
```

```
  0.3536 -0.3536 lineto
  %middle circumferential
  0.3536 -0.3536 lineto
  0.413 -0.2818 lineto
  0.4594 -0.1974 lineto
  0.4894 -0.1024 lineto
  0.5 0.0 lineto
  %end circumferential
  0.5 0.0 lineto
  0.375 0.0 lineto
  %middle radial
  0.375 0.0 lineto
  0.25 0.0 lineto
  %end radial
  0.25 0.0 lineto
  0.2297 -0.0987 lineto
  0.1768 -0.1768 lineto
  %middle circumferential
  0.1768 -0.1768 lineto
  0.0922 -0.2324 lineto
  0.0 -0.25 lineto
  %end circumferential
  closepath
currentgray  0.45 setgray  fill  setgray

    [2598 lines omitted]
% Vertical line from (0.0, -0.5) to (0.0, 0.0)
newpath
  0.0 -0.5 moveto
  0.0 0.0 lineto
0.05 setlinewidth  1 setlinecap  stroke
% Vertical line from (0.0, -0.5) to (0.0, -1.0)
newpath
  0.0 -0.5 moveto
  0.0 -1.0 lineto
0.05 setlinewidth  1 setlinecap  stroke
% Vertical line from (0.0, -2.0) to (0.0, -1.0)
newpath
  0.0 -2.0 moveto
  0.0 -1.0 lineto
```

```
0.05 setlinewidth   1 setlinecap  stroke
% Vertical line from (0.0, -2.0) to (0.0, -1.1579208923731617E77)
newpath
  0.0 -2.0 moveto
  0.0 -6.3553 lineto
  0.0 -6.378103166302659 lineto
  0.0 -6.378103166302659 lineto
  0.0 -6.378103166302659 lineto
0.05 setlinewidth 1 setlinecap stroke


    [84 lines omitted]
% End of PostScript file for plot of function IDENTITY
```

Here is the program that generated the PostScript code for the graphs shown in figures 12.1 through 12.20. It contains a mixture of fairly general mechanisms and *ad hoc* kludges for plotting functions of a single complex argument while gracefully handling extremely large and small values, branch cuts, singularities, and periodic behavior. The aim was to provide a simple user interface that would not require the caller to provide special advice for each function to be plotted. The file for figure 12.1, for example, was generated by the call `(picture 'identity)`, which resulted in the writing of a file named `identity-plot.ps`.

The program assumes that any periodic behavior will have a period that is a multiple of 2π; that branch cuts will fall along the real or imaginary axis; and that singularities or very large or small values will occur only at the origin, at ± 1 or $\pm i$, or on the boundaries of the annuli (particularly those with radius $\pi/2$ or π). The central function is `parametric-path`, which accepts four arguments: two real numbers that are the endpoints of an interval of real numbers, a function that maps this interval into a path in the complex plane, and the function to be plotted; the task of `parametric-path` is to generate PostScript code (a series of `lineto` operations) that will plot an approximation to the image of the parametric path as transformed by the function to be plotted. Each of the functions `hline`, `vline`, `-hline`, `-vline`, `radial`, and `circumferential` takes appropriate parameters and returns a function suitable for use as the third argument to `parametric-path`. There is some code that defends against errors (by using `ignore-errors`) and against certain peculiarities of IEEE floating-point arithmetic (the code that checks for not-a-number (NaN) results).

The program is offered here without further comment or apology.

```
(defparameter units-to-show 4.1)
(defparameter text-width-in-picas 28.0)
(defparameter device-pixels-per-inch 300)
(defparameter pixels-per-unit
  (* (/ (/ text-width-in-picas 6)
        (* units-to-show 2))
     device-pixels-per-inch))

(defparameter big (sqrt (sqrt most-positive-single-float)))
(defparameter tiny (sqrt (sqrt least-positive-single-float)))

(defparameter path-really-losing 1000.0)
(defparameter path-outer-limit (* units-to-show (sqrt 2) 1.1))
(defparameter path-minimal-delta (/ 10 pixels-per-unit))
(defparameter path-outer-delta (* path-outer-limit 0.3))
(defparameter path-relative-closeness 0.00001)
(defparameter back-off-delta 0.0005)
```

```
(defun comment-line (stream &rest stuff)
  (format stream "~%% ")
  (apply #'format stream stuff)
  (format t "~%% ")
  (apply #'format t stuff))

(defun parametric-path (from to paramfn plotfn)
  (assert (and (plusp from) (plusp to)))
  (flet ((domainval (x) (funcall paramfn x))
         (rangeval (x) (funcall plotfn (funcall paramfn x)))
         (losing (x) (or (null x)
                         (/= (realpart x) (realpart x))  ;NaN?
                         (/= (imagpart x) (imagpart x))  ;NaN?
                         (> (abs (realpart x)) path-really-losing)
                         (> (abs (imagpart x)) path-really-losing))))
    (when (> to 1000.0)
      (let ((f0 (rangeval from))
            (f1 (rangeval (+ from 1)))
            (f2 (rangeval (+ from (* 2 pi))))
            (f3 (rangeval (+ from 1 (* 2 pi))))
            (f4 (rangeval (+ from (* 4 pi)))))
        (flet ((close (x y)
                 (or (< (careful-abs (- x y)) path-minimal-delta)
                     (< (careful-abs (- x y))
                        (* (+ (careful-abs x) (careful-abs y))
                           path-relative-closeness)))))
          (when (and (close f0 f2)
                     (close f2 f4)
                     (close f1 f3)
                     (or (and (close f0 f1)
                              (close f2 f3))
                         (and (not (close f0 f1))
                              (not (close f2 f3)))))
            (format t "~&Periodicity detected.")
            (setq to (+ from (* (signum (- to from)) 2 pi)))))))))
```

```
(let ((fromrange (ignore-errors (rangeval from)))
      (torange (ignore-errors (rangeval to))))
  (if (losing fromrange)
      (if (losing torange)
          '()
          (parametric-path (back-off from to) to paramfn plotfn))
      (if (losing torange)
          (parametric-path from (back-off to from) paramfn plotfn)
          (expand-path (refine-path (list from to) #'rangeval)
                       #'rangeval))))))
```

```
(defun back-off (point other)
  (if (or (> point 10.0) (< point 0.1))
      (let ((sp (sqrt point)))
        (if (or (> point sp other) (< point sp other))
            sp
            (* sp (sqrt other))))
      (+ point (* (signum (- other point)) back-off-delta))))

(defun careful-abs (z)
  (cond ((or (> (realpart z) big)
             (< (realpart z) (- big))
             (> (imagpart z) big)
             (< (imagpart z) (- big)))
         big)
        ((complexp z) (abs z))
        ((minusp z) (- z))
        (t z)))
```

```
(defparameter max-refinements 5000)

(defun refine-path (original-path rangevalfn)
  (flet ((rangeval (x) (funcall rangevalfn x)))
    (let ((path original-path))
      (do ((j 0 (+ j 1)))
          ((null (rest path)))
        (when (zerop (mod (+ j 1) max-refinements))
              (break "Runaway path"))
        (let* ((from (first path))
               (to (second path))
               (fromrange (rangeval from))
               (torange (rangeval to))
               (dist (careful-abs (- torange fromrange)))
               (mid (* (sqrt from) (sqrt to)))
               (midrange (rangeval mid)))
```

```
        (cond ((or (and (far-out fromrange) (far-out torange))
                   (and (< dist path-minimal-delta)
                        (< (abs (- midrange fromrange))
                           path-minimal-delta)
                        ;; Next test is intentionally asymmetric to
                        ;;  avoid problems with periodic functions.
                        (< (abs (- (rangeval (/ (+ to (* from 1.5))
                                                2.5))
                                   fromrange))
                           path-minimal-delta)))
               (pop path))
              ((= mid from) (pop path))
              ((= mid to) (pop path))
              (t (setf (rest path) (cons mid (rest path))))))))
  original-path)

(defun expand-path (path rangevalfn)
  (flet ((rangeval (x) (funcall rangevalfn x)))
    (let ((final-path (list (rangeval (first path)))))
      (do ((p (rest path) (cdr p)))
          ((null p)
           (unless (rest final-path)
             (break "Singleton path"))
           (reverse final-path))
        (let ((v (rangeval (car p))))
          (cond ((and (rest final-path)
                      (not (far-out v))
                      (not (far-out (first final-path)))
                      (between v (first final-path)
                               (second final-path)))
                 (setf (first final-path) v))
                ((null (rest p))   ;Mustn't omit last point
                 (push v final-path))
                ((< (abs (- v (first final-path))) path-minimal-delta))
```

```
            ((far-out v)
             (unless (and (far-out (first final-path))
                          (< (abs (- v (first final-path)))
                             path-outer-delta))
               (push (* 1.01 path-outer-limit (signum v))
                     final-path)))
            (t (push v final-path))))))))

(defun far-out (x)
  (> (careful-abs x) path-outer-limit))

(defparameter between-tolerance 0.000001)

(defun between (p q r)
  (let ((px (realpart p)) (py (imagpart p))
        (qx (realpart q)) (qy (imagpart q))
        (rx (realpart r)) (ry (imagpart r)))
    (and (or (<= px qx rx) (>= px qx rx))
         (or (<= py qy ry) (>= py qy ry))
         (< (abs (- (* (- qx px) (- ry qy))
                    (* (- rx qx) (- qy py))))
            between-tolerance))))
```

```
(defun circle (radius)
  #'(lambda (angle) (* radius (cis angle))))

(defun hline (imag)
  #'(lambda (real) (complex real imag)))

(defun vline (real)
  #'(lambda (imag) (complex real imag)))

(defun -hline (imag)
  #'(lambda (real) (complex (- real) imag)))

(defun -vline (real)
  #'(lambda (imag) (complex real (- imag))))

(defun radial (phi quadrant)
  #'(lambda (rho) (repair-quadrant (* rho (cis phi)) quadrant)))

(defun circumferential (rho quadrant)
  #'(lambda (phi) (repair-quadrant (* rho (cis phi)) quadrant)))

;;; Quadrant is 0, 1, 2, or 3, meaning I, II, III, or IV.

(defun repair-quadrant (z quadrant)
  (complex (* (+ (abs (realpart z)) tiny)
              (case quadrant (0 1.0) (1 -1.0) (2 -1.0) (3 1.0)))
           (* (+ (abs (imagpart z)) tiny)
              (case quadrant (0 1.0) (1 1.0) (2 -1.0) (3 -1.0)))))

(defun clamp-real (x)
  (if (far-out x)
      (* (signum x) path-outer-limit)
      (round-real x)))

(defun round-real (x)
  (/ (round (* x 10000.0)) 10000.0))

(defun round-point (z)
  (complex (round-real (realpart z)) (round-real (imagpart z))))
```

```
(defparameter hiringshade 0.97)
(defparameter loringshade 0.45)

(defparameter ticklength 0.12)
(defparameter smallticklength 0.09)
```

```
;;; This determines the pattern of lines and annuli to be drawn.
(defun moby-grid (&optional (fn 'sqrt) (stream t))
  (comment-line stream "Moby grid for function ~S" fn)
  (shaded-annulus 0.25 0.5 4 hiringshade loringshade fn stream)
  (shaded-annulus 0.75 1.0 8 hiringshade loringshade fn stream)
  (shaded-annulus (/ pi 2) 2.0 16 hiringshade loringshade fn stream)
  (shaded-annulus 3 pi 32 hiringshade loringshade fn stream)
  (moby-lines :horizontal 1.0 fn stream)
  (moby-lines :horizontal -1.0 fn stream)
  (moby-lines :vertical 1.0 fn stream)
  (moby-lines :vertical -1.0 fn stream)
  (let ((tickline 0.015)
        (axisline 0.008))
    (flet ((tick (n) (straight-line (complex n ticklength)
                                    (complex n (- ticklength))
                                    tickline
                                    stream))
           (smalltick (n) (straight-line (complex n smallticklength)
                                         (complex n (- smallticklength))
                                         tickline
                                         stream)))
      (comment-line stream "Real axis")
      (straight-line #c(-5 0) #c(5 0) axisline stream)
      (dotimes (j (floor units-to-show))
        (let ((q (+ j 1))) (tick q) (tick (- q))))
      (dotimes (j (floor units-to-show (/ pi 2)))
        (let ((q (* (/ pi 2) (+ j 1))))
          (smalltick q)
          (smalltick (- q)))))
    (flet ((tick (n) (straight-line (complex ticklength n)
                                    (complex (- ticklength) n)
                                    tickline
                                    stream))
           (smalltick (n) (straight-line (complex smallticklength n)
                                         (complex (- smallticklength) n)
                                         tickline
                                         stream)))
```

```
(comment-line stream "Imaginary axis")
(straight-line #c(0 -5) #c(0 5) axisline stream)
(dotimes (j (floor units-to-show))
  (let ((q (+ j 1))) (tick q) (tick (- q))))
(dotimes (j (floor units-to-show (/ pi 2)))
  (let ((q (* (/ pi 2) (+ j 1))))
    (smalltick q)
    (smalltick (- q)))))))
```

```
(defun straight-line (from to wid stream)
  (format stream
          "~%newpath  ~S ~S moveto  ~S ~S lineto  ~S ~
           setlinewidth  1  setlinecap  stroke"
          (realpart from)
          (imagpart from)
          (realpart to)
          (imagpart to)
          wid))



;;; This function draws the lines for the pattern.
(defun moby-lines (orientation signum plotfn stream)
  (let ((paramfn (ecase orientation
                   (:horizontal (if (< signum 0) #'-hline #'hline))
                   (:vertical (if (< signum 0) #'-vline #'vline)))))
    (flet ((foo (from to other wid)
             (ecase orientation
               (:horizontal
                (comment-line stream
                              "Horizontal line from (~S, ~S) to (~S, ~S)"
                              (round-real (* signum from))
                              (round-real other)
                              (round-real (* signum to))
                              (round-real other)))
               (:vertical
                (comment-line stream
                              "Vertical line from (~S, ~S) to (~S, ~S)"
                              (round-real other)
                              (round-real (* signum from))
                              (round-real other)
                              (round-real (* signum to)))))
             (postscript-path
               stream
               (parametric-path from
                                to
                                (funcall paramfn other)
                                plotfn))
             (postscript-penstroke stream wid)))
```

```
    (let* ((thick 0.05)
           (thin 0.02))
      ;; Main axis
      (foo 0.5 tiny 0.0 thick)
      (foo 0.5 1.0 0.0 thick)
      (foo 2.0 1.0 0.0 thick)
      (foo 2.0 big 0.0 thick)
      ;; Parallels at 1 and -1
      (foo 2.0 tiny 1.0 thin)
      (foo 2.0 big 1.0 thin)
      (foo 2.0 tiny -1.0 thin)
      (foo 2.0 big -1.0 thin)
      ;; Parallels at 2, 3, -2, -3
      (foo tiny big 2.0 thin)
      (foo tiny big -2.0 thin)
      (foo tiny big 3.0 thin)
      (foo tiny big -3.0 thin)))))

(defun splice (p q)
  (let ((v (car (last p)))
        (w (first q)))
    (and (far-out v)
         (far-out w)
         (>= (abs (- v w)) path-outer-delta)
         ;; Two far-apart far-out points.  Try to walk around
         ;;  outside the perimeter, in the shorter direction.
         (let* ((pdiff (phase (/ v w)))
                (npoints (floor (abs pdiff) (asin .2)))
                (delta (/ pdiff (+ npoints 1)))
                (incr (cis delta)))
           (do ((j 0 (+ j 1))
                (p (list w "end splice") (cons (* (car p) incr) p)))
               ((= j npoints) (cons "start splice" p)))))))
```

;;; This function draws the annuli for the pattern.

```
(defun shaded-annulus (inner outer sectors firstshade lastshade fn stream)
  (assert (zerop (mod sectors 4)))
  (comment-line stream "Annulus ~S ~S ~S ~S ~S"
                (round-real inner) (round-real outer)
                sectors firstshade lastshade)
  (dotimes (jj sectors)
    (let ((j (- sectors jj 1)))
      (let* ((lophase (+ tiny (* 2 pi (/ j sectors))))
             (hiphase (* 2 pi (/ (+ j 1) sectors)))
             (midphase (/ (+ lophase hiphase) 2.0))
             (midradius (/ (+ inner outer) 2.0))
             (quadrant (floor (* j 4) sectors)))
        (comment-line stream "Sector from ~S to ~S (quadrant ~S)"
                      (round-real lophase)
                      (round-real hiphase)
                      quadrant)
        (let ((p0 (reverse (parametric-path midradius
                                            inner
                                            (radial lophase quadrant)
                                            fn)))
              (p1 (parametric-path midradius
                                   outer
                                   (radial lophase quadrant)
                                   fn))
              (p2 (reverse (parametric-path midphase
                                            lophase
                                            (circumferential outer
                                                             quadrant)
                                            fn)))
              (p3 (parametric-path midphase
                                   hiphase
                                   (circumferential outer quadrant)
                                   fn))
              (p4 (reverse (parametric-path midradius
                                            outer
                                            (radial hiphase quadrant)
                                            fn)))
```

```
    (p5 (parametric-path midradius
                         inner
                         (radial hiphase quadrant)
                         fn))
    (p6 (reverse (parametric-path midphase
                                  hiphase
                                  (circumferential inner
                                                   quadrant)
                                  fn)))
    (p7 (parametric-path midphase
                         lophase
                         (circumferential inner quadrant)
                         fn)))
(postscript-closed-path stream
  (append
    p0 (splice p0 p1) '("middle radial")
    p1 (splice p1 p2) '("end radial")
    p2 (splice p2 p3) '("middle circumferential")
    p3 (splice p3 p4) '("end circumferential")
    p4 (splice p4 p5) '("middle radial")
    p5 (splice p5 p6) '("end radial")
    p6 (splice p6 p7) '("middle circumferential")
    p7 (splice p7 p0) '("end circumferential")
    )))
```

```
      (postscript-shade stream
                        (/ (+ (* firstshade (- (- sectors 1) j))
                              (* lastshade j))
                           (- sectors 1)))))))

(defun postscript-penstroke (stream wid)
  (format stream "~%~S setlinewidth   1 setlinecap  stroke"
          wid))

(defun postscript-shade (stream shade)
  (format stream "~%currentgray   ~S setgray   fill   setgray"
          shade))

(defun postscript-closed-path (stream path)
  (unless (every #'far-out (remove-if-not #'numberp path))
    (postscript-raw-path stream path)
    (format stream "~%  closepath")))

(defun postscript-path (stream path)
  (unless (every #'far-out (remove-if-not #'numberp path))
    (postscript-raw-path stream path)))

;;; Print a path as a series of PostScript "lineto" commands.
(defun postscript-raw-path (stream path)
  (format stream "~%newpath")
  (let ((fmt "~%  ~S ~S moveto"))
    (dolist (pt path)
      (cond ((stringp pt)
             (format stream "~%  %~A" pt))
            (t (format stream
                       fmt
                       (clamp-real (realpart pt))
                       (clamp-real (imagpart pt)))
               (setq fmt "~%  ~S ~S lineto"))))))

;;; Definitions of functions to be plotted that are not
;;; standard Common Lisp functions.

(defun one-plus-over-one-minus (x) (/ (+ 1 x) (- 1 x)))
```

```
(defun one-minus-over-one-plus (x) (/ (- 1 x) (+ 1 x)))

(defun sqrt-square-minus-one (x) (sqrt (- 1 (* x x))))

(defun sqrt-one-plus-square (x) (sqrt (+ 1 (* x x))))
```

```
;;; Because X3J13 voted for a new definition of the atan function,
;;; the following definition was used in place of the atan function
;;; provided by the Common Lisp implementation I was using.

(defun good-atan (x)
  (/ (- (log (+ 1 (* x #c(0 1))))
        (log (- 1 (* x #c(0 1)))))
     #c(0 2)))

;;; Because the first edition had an erroneous definition of atanh,
;;; the following definition was used in place of the atanh function
;;; provided by the Common Lisp implementation I was using.

(defun really-good-atanh (x)
  (/ (- (log (+ 1 x))
        (log (- 1 x)))
     2))



;;; This is the main procedure that is intended to be called by a user.
(defun picture (&optional (fn #'sqrt))
  (with-open-file (stream (concatenate 'string
                                       (string-downcase (string fn))
                                       "-plot.ps")
                          :direction :output)
    (format stream "% PostScript file for plot of function ~S~%" fn)
    (format stream "% Plot is to fit in a region ~S inches square~%"
            (/ text-width-in-picas 6.0))
    (format stream
            "%  showing axes extending ~S units from the origin.~%"
            units-to-show)
    (let ((scaling (/ (* text-width-in-picas 12) (* units-to-show 2))))
      (format stream "~%~S ~:*~S scale" scaling))
    (format stream "~%~S ~:*~S translate" units-to-show)
    (format stream "~%newpath")
    (format stream "~%  ~S ~S moveto" (- units-to-show) (- units-to-show))
    (format stream "~%  ~S ~S lineto" units-to-show (- units-to-show))
    (format stream "~%  ~S ~S lineto" units-to-show units-to-show)
    (format stream "~%  ~S ~S lineto" (- units-to-show) units-to-show)
```

```
(format stream "~%  closepath")
(format stream "~%clip")
(moby-grid fn stream)
(format stream
      "~%% End of PostScript file for plot of function ~S"
      fn)
(terpri stream)))
```

12.6 Приведение типов и доступ к компонентам чисел

Тогда как большинство арифметических функций будут оперировать любым типом чисел, выполняя при необходимости приведения, следующие функции позволяют явно преобразовывать типы данных.

[Функция] **float** *number* **&optional** *other*

Преобразует любое некомплексное число в число с плавающей точкой. При отсутствии необязательного параметра, если *number* уже является числом с плавающей точкой, то оно и будет возвращено, иначе число будет преобразовано в `single-float`. Если аргумент *other* указан, тогда он должен быть числом с плавающей точкой, и *number* будет конвертирован в такой же формат как у *other*.

Смотрите также `coerce`.

[Функция] **rational** *number*
[Функция] **rationalize** *number*

Каждая из этих функций преобразует любое некомплексное число в рациональное. Если аргумент уже является рациональным, он возвращается как есть. Две функции различаются в том, как они обрабатывают числа с плавающей точкой.

`rational` предполагает, что число с плавающей точкой совершенно точно, и возвращает рациональное число математически эквивалентное значению числа с плавающей точкой.

`rationalize` предполагает, что число с плавающей точкой приближенное, и может возвращать любое рациональное число, для которого исходное число является наилучшим приближением. Функция пытается сохранить числитель и знаменатель наименьшими насколько это возможно.

Следующие тождества всегда справедливы

(float (rational x) x) $\equiv x$

и

(float (rationalize x) x) $\equiv x$

То есть, преобразование числа с плавающей точкой любым из методов туда и обратно даёт исходное число. Различие в том, что `rational` обычно имеет более простую недорогую реализацию, тогда как `rationalize` представляет более «красивое» число.

[Функция] **numerator** *rational*
[Функция] **denominator** *rational*

Эти функции принимают рациональное число (целое или дробное) и возвращают в качестве целого числа числитель или знаменатель дроби, приведённое к каноническому виду. Числитель целого числа и является этим числом. Знаменатель целого числа `1`. Следует отметить, что

(gcd (numerator x) (denominator x)) $\Rightarrow$ 1

Знаменатель будет всегда строго положительным числом. Числитель может быть любым целым числом. Например:

(numerator (/ 8 -6)) $\Rightarrow$ -4
(denominator (/ 8 -6)) $\Rightarrow$ 3

В Common Lisp'е нет функции `fix`, потому что есть несколько интересных способов преобразовать нецелое число к целому. Эти способы представлены функциями ниже, которые выполняются не только преобразование типа, но также некоторые нетривиальные вычисления.

[Функция] **floor** *number* **&optional** *divisor*
[Функция] **ceiling** *number* **&optional** *divisor*
[Функция] **truncate** *number* **&optional** *divisor*
[Функция] **round** *number* **&optional** *divisor*

При вызове с одним аргументом, каждая из этих функций преобразует аргумент *number* (который не может быть комплексным числом) в целое число. Если аргумент уже является целым числом, то он немедленно возвращается в качестве результата. Если аргумент дробь или число с плавающей точкой, для конвертации функции используют различные алгоритмы.

`floor` преобразовывает аргумент путём отсечения к отрицательной бесконечности, то есть, результатом является наибольшее целое число, которое не больше чем аргумент.

`ceiling` преобразовывает аргумент путём отсечения к положительной бесконечности, то есть, результатом является наименьшее целое число, которое не меньше чем аргумент.

`ceiling` преобразовывает аргумент путём отсечения к нулю, то есть, результатом является целое число с таким же знаком, которое имеет наибольшую целую величину, но не большую чем аргумент.

Следующая таблица содержит то, что возвращают четыре функции для разных аргументов.

Аргумент	floor	ceiling	truncate	round
2.6	2	3	2	3
2.5	2	3	2	2
2.4	2	3	2	2
0.7	0	1	0	1
0.3	0	1	0	0
-0.3	-1	0	0	0
-0.7	-1	0	0	-1
-2.4	-3	-2	-2	-2
-2.5	-3	-2	-2	-2
-2.6	-3	-2	-2	-3

Если указан второй аргумент *divisor*, тогда аргумент *number* будет разделен на *divisor*, а затем уже будут проведены вышеописанные действия. Например, `(floor 5 2)` ≡ `(values (floor (/ 5 2)))`, но первый вариант потенциально эффективнее.

This statement is not entirely accurate; one should instead say that `(values (floor 5 2))` ≡ `(values (floor (/ 5 2)))`, because there is a second value to consider, as discussed below. In other words, the first values returned by the two forms will be the same, but in general the second values will differ. Indeed, we have

(floor 5 2) ⇒ 2 and 1
(floor (/ 5 2)) ⇒ 2 and 1/2

for this example.

divisor может любым числом, кроме комплексного. *divisor* не может равняться нулю. Случай вызова функции с одним аргументом эквивалентен вызову с двумя аргументами, последний из которых равен `1`.

In other words, the one-argument case returns an integer and fractional part for the *number*: `(truncate 5.3)` ⇒ `5.0` and `0.3`, for example.

Каждая из функций возвращает *два* значения. Второе значение является остатком и может быть получено с помощью `multiple-value-bind` или другими подобными конструкциями. Если любая из этих функций получает два аргумента x и y и возвращает q и r, тогда $q \cdot y + r = x$. Первое значение результата q всегда целочисленное. Остаток r целочисленный, если оба аргумента были целочисленными, и

рациональное, если оба аргумента были рациональными, и с плавающей точкой, если один из аргументов был с плавающей точкой. Если при вызове был указан один аргумент, то тип данных остатка всегда такой же как и этот аргумент.

Когда указывается только один аргумент, сумма двух значений результата точно равняется переданному в параметре значению.

[Функция] **mod** *number divisor*
[Функция] **rem** *number divisor*

`mod` выполняет операцию `floor` для двух аргументов и возвращает *второй* результат `floor`. Таким же образом, `rem` выполняет операцию `truncate` для двух аргументов и возвращает *второй* результат `truncate`.

Таким образом `mod` и `rem` являются обычными функциями вычисления остатка от деления двух чисел. Аргументы могут быть также числами с плавающей точкой.

(mod 13 4) ⇒ 1
(mod -13 4) ⇒ 3
(mod 13 -4) ⇒ -3
(mod -13 -4) ⇒ -1
(mod 13.4 1) ⇒ 0.4
(mod -13.4 1) ⇒ 0.6

(rem 13 4) ⇒ 1
(rem -13 4) ⇒ -1
(rem 13 -4) ⇒ 1
(rem -13 -4) ⇒ -1
(rem 13.4 1) ⇒ 0.4
(rem -13.4 1) ⇒ -0.4

[Функция] **ffloor** *number* **&optional** *divisor*
[Функция] **fceiling** *number* **&optional** *divisor*
[Функция] **ftruncate** *number* **&optional** *divisor*
[Функция] **fround** *number* **&optional** *divisor*

Эти функции похожи на `floor`, `ceiling`, `truncate` и `round` за исключением того, что результат (первый из двух) всегда целое число, а не число с плавающей точкой. Это примерно, как если бы `ffloor` передала аргументы в `floor`, а затем применила к первому результату `float` и вернула полученную пару значений. Однако, на практике `ffloor` может быть реализована более эффективно. Такое же описание подходит к остальным трём функциям. Если первый аргумент является числом с плавающей точкой, и второй аргумент не точнее типа первого, тогда первый результат будет такого же типа как первый аргумент. Например:

(ffloor -4.7) ⇒ -5.0 and 0.3
(ffloor 3.5d0) ⇒ 3.0d0 and 0.5d0

[Функция] **decode-float** *float*
[Функция] **scale-float** *float integer*
[Функция] **float-radix** *float*
[Функция] **float-sign** *float1* **&optional** *float2*
[Функция] **float-digits** *float*
[Функция] **float-precision** *float*
[Функция] **integer-decode-float** *float*

Функция `decode-float` принимает числа с плавающей точкой и возвращает три значения.

Первое значение является мантиссой и числом того же типа, что и аргумент. Второе значение является целочисленной экспонентой. Третье значение отображает знак аргумента (-1.0 или 1.0) и является и числом того же типа, что и аргумент . Пусть b есть система счисления для отображения чисел с плавающей точкой, тогда `decode-float` делит аргумент на b в некоторой степени, чтобы привести значение в промежуток включая $1/b$ и не включая 1 и возвращает частное в качестве первого значения FIXME. Однако, если аргумент равен нулю результат равен абсолютному значению аргумента (то есть, если существует отрицательный ноль, то для него возвращается положительный ноль).

Второе значение `decode-float` является целочисленным экспонентой e, которая равняется степени, в которую было возведено b. Если аргумент равен нулю, то может быть возвращено любое целое число, при условии, что тожество, описанное ниже для `scale-format`, имеет место быть.

Третье значение `decode-float` является числом с плавающей точкой в том же формате, что и аргумент, абсолютное значение которого равно 1, и знак совпадает со знаком аргумента.

Функция `scale-float` принимает число с плавающей точкой f (не обязательно между $1/b$ и 1) и целое число k, и возвращает `(* f (expt (float b f) k))`. (Использование `scale-float` может быть более эффективным, чем использование возведения в степень или умножения и позволяет избежать переполнений).

Следует отметить, что

(multiple-value-bind (signif expon sign)
(decode-float f)
(scale-float signif expon))
$\equiv$ (abs f)

и

(multiple-value-bind (signif expon sign)
(decode-float f)
(* (scale-float signif expon) sign))
$\equiv f$

Функция `float-radix` возвращает (в качестве целого числа) основание b для числа с плавающей точкой.

Функция `float-sign` возвращает такое число с плавающей точкой z, что z и *float1* имеют одинаковый знак, и z и *float2* имеют равное абсолютное значение. Аргумент *float2* по-умолчанию имеет значение `(float 1 float1)`. Таким образом `(float-sign x)` всегда возвращает `1.0` или `-1.0` в таком же формате и с тем же знаком, что и x. (Следует отметить, что если реализация содержит различные представления для отрицательного и положительного нулей, тогда `(float-sign -0.0)` $\Rightarrow$ `-1.0`.)

Функция `float-digits` возвращает целочисленное количество цифр используемых в представлении аргумента. Функция `float-precision` возвращает целочисленное количество цифр в мантиссе аргумента. Если аргумент равен нулю, то результатом также будет ноль. Для нормализованных чисел с плавающей точкой, результаты `float-digits` и `float-precision` будут такими же. Но в случае с денормализованными числами или нулём точность будет меньше чем количество цифр в представлении.

Функция `integer-decode-float` похожа на `decode-float`, но в качестве первого значение возвращает масштабированную целочисленную мантиссу. Для аргумента f, это число будет строго меньше чем

```
(expt b (float-precision f))
```

на не менее чем

```
(expt b (- (float-precision f) 1))
```

за исключением того, что если f равно нулю, тогда целочисленное значение также будет равно нулю.

Второе значение имеет такую же связь с первым значением, как и для `decode-float`:

(multiple-value-bind (signif expon sign)
(integer-decode-float f)
(scale-float (float signif f) expon))
≡ (abs f)

Третье значение `integer-decode-float` будет `1` или `-1`.

Обоснование: Эти функции позволяют писать машиннонезависиммые или как минимум машиннопараметризированные приложения с вычислениями чисел с плавающими точками с необходимой эффективностью.

[Функция] **complex** *realpart* **&optional** *imagpart*

Аргументы должны быть некомплексными числами. Результатом является число, которое имеет действительную часть *realpart* и мнимую *imagpart*, возможно конвертированные к одному типу. Если *imagpart* не указаны, тогда используется `(coerce 0 (type-of realpart))`. Необходимо отметить, что если обе части комплексного число рациональны, и мнимая часть равна нуля, то результатом будет только действительная часть *realpart* так как в силу вступят правила канонизации. Таким образом результат `complex` не всегда является комплексным числом. Он может быть просто рациональным числом (`rational`).

[Функция] **realpart** *number*
[Функция] **imagpart** *number*

Эти функции возвращают действительную и мнимую части комплексного числа. Если *number* не является комплексным числом,

тогда `realpart` возвращает это число, а `imagpart` возвращает `(* 0 number)`, другими словами, `0` того типа, каким был аргумент.

A clever way to multiply a complex number z by i is to write

(complex (- (imagpart z)) (realpart z))

instead of `(* z #c(0 1))`. This cleverness is not always gratuitous; it may be of particular importance in the presence of minus zero. For example, if we are using IEEE standard floating-point arithmetic and $z = 4 + 0i$, the result of the clever expression is $-0 + 4i$, a true 90° rotation of z, whereas the result of `(* z #c(0 1))` is likely to be

$$\begin{aligned}(4 + 0i)(+0 + i) &= ((4)(+0) - (+0)(1)) + ((4)(1) + (+0)(+0))i \\ &= ((+0) - (+0)) + ((4) + (+0))i = +0 + 4i\end{aligned}$$

which could land on the wrong side of a branch cut, for example.

12.7 Логические операции над числами

Логические операции в данном разделе в качестве аргументов требуют целых чисел. Передача числа любого другого формата является ошибкой. Функции обрабатывают целые числа, как если бы они были представлены в двух системах счисления. FIXME

Заметка для реализации: Внутренне, конечно, реализация Common Lisp'а может использовать и может и нет представление числа с дополнительным кодом. Все, что необходимо это чтобы логические операции выполняли вычисление так, как описано в разделе.

Логические операции предоставляют удобный способ для представления бесконечного вектора битов. Пусть такой концептуальный вектор будет индексироваться с помощью неотрицательного целого. Тогда биту j присваивается «вес (weight)» 2^j. Предположим, что лишь конечное число битов являются 1 или только конечное число битов являются 0. Вектор, у которого конечное число битов 1, представлен как сумма всех весов этих битов, и является положительным числом. Вектор, у которого конечное число битов 0,

представлен как `-1` минус сумма всех весов этих битов, и является отрицательным числом. FIXME

Данный метод использования целых чисел для представления битовых векторов может в свою очередь использоваться для представления множеств. Предположим, что некоторая (возможно бесконечная) совокупность рассуждений для множеств отображается в неотрицательные целые числа. FIXME Тогда множество может быть представлено как битовый вектор. Элемент принадлежит множеству, если бит, индекс которого соответствует элементу, является 1. Таким образом все конечные множества могут быть представлены с помощью положительных целых, и множества, дополнения которых конечны, с помощью отрицательных чисел. Функции `logior`, `logand` и `logxor`, определённые ниже, вычисляют объединение, пересечение и симметричную разность для таких множеств.

[Функция] **logior** **&rest** *integers*

Функция возвращает побитовое логическое *или* для аргументов. Если не задан ни один аргумент, возвращается ноль.

[Функция] **logxor** **&rest** *integers*

Функция возвращает побитовое логическое *исключающее или* для аргументов. Если не задан ни один аргумент, возвращается ноль.

[Функция] **logand** **&rest** *integers*

Функция возвращает побитовое логическое *и* для аргументов. Если ни один аргумент не задан, возвращается ноль.

[Функция] **logeqv** **&rest** *integers*

Функция возвращает побитовую логическую *эквивалентность* (также известную как *исключающее отрицающее или* для аргументов. Если не задан ни один аргумент, возвращается ноль.

[Функция] **lognand** *integer1 integer2*
[Функция] **lognor** *integer1 integer2*
[Функция] **logandc1** *integer1 integer2*
[Функция] **logandc2** *integer1 integer2*
[Функция] **logorc1** *integer1 integer2*
[Функция] **logorc2** *integer1 integer2*

Данные функции служат для шести нетривиальные битовый логический операций для двух аргументов. Так как они не ассоциативны, они принимают только два аргумента.

(lognand *n1 n2*) ≡ (lognot (logand *n1 n2*))
(lognor *n1 n2*) ≡ (lognot (logior *n1 n2*))
(logandc1 *n1 n2*) ≡ (logand (lognot *n1*) *n2*)
(logandc2 *n1 n2*) ≡ (logand *n1* (lognot *n2*))
(logorc1 *n1 n2*) ≡ (logior (lognot *n1*) *n2*)
(logorc2 *n1 n2*) ≡ (logior *n1* (lognot *n2*))

В следующей таблице перечислены десять битовых логических операций для двух целых чисел.

integer1	0	0	1	1	
integer2	0	1	0	1	Имя операции
logand	0	0	0	1	и
logior	0	1	1	1	или
logxor	0	1	1	0	исключающее или
logeqv	1	0	0	1	эквивалентность (исключающее отрицающее или)
lognand	1	1	1	0	не и
lognor	1	0	0	0	не или
logandc1	0	1	0	0	не *integer1* и *integer2*
logandc2	0	0	1	0	*integer1* и не *integer2*
logorc1	1	1	0	1	не *integer1* или *integer2*
logorc2	1	0	1	1	*integer1* или не *integer2*

[Функция] **boole** *op integer1 integer2*
[Константа] **boole-clr**
[Константа] **boole-set**
[Константа] **boole-1**
[Константа] **boole-2**
[Константа] **boole-c1**
[Константа] **boole-c2**
[Константа] **boole-and**
[Константа] **boole-ior**
[Константа] **boole-xor**
[Константа] **boole-eqv**
[Константа] **boole-nand**
[Константа] **boole-nor**
[Константа] **boole-andc1**
[Константа] **boole-andc2**
[Константа] **boole-orc1**
[Константа] **boole-orc2**

Функция `boole` принимает операцию *op* и два целых числа, и возвращает целое число полученное применением операции *op* к этим двум числам. Точные значения шестнадцати констант зависят от реализации, но они подходят для использования в качестве первого аргумента для `boole`:

integer1	0	0	1	1	
integer2	0	1	0	1	Выполняемая операция
boole-clr	0	0	0	0	всегда 0
boole-set	1	1	1	1	всегда 1
boole-1	0	0	1	1	*integer1*
boole-2	0	1	0	1	*integer2*
boole-c1	1	1	0	0	дополнение *integer1*
boole-c2	1	0	1	0	дополнение *integer2*
boole-and	0	0	0	1	и
boole-ior	0	1	1	1	или
boole-xor	0	1	1	0	исключающее или
boole-eqv	1	0	0	1	эквивалентность (исключительное отрицающее или)
boole-nand	1	1	1	0	не и
boole-nor	1	0	0	0	не или
boole-andc1	0	1	0	0	не *integer1* и *integer2*
boole-andc2	0	0	1	0	*integer1* и не *integer2*
boole-orc1	1	1	0	1	не *integer1* или *integer2*
boole-orc2	1	0	1	1	*integer1* или не *integer2*

Таким образом `boole` может вычислять все шестнадцать логических функций для двух аргументов. В целом,

(boole boole-and x y) ≡ (logand x y)

и далее по аналогии. `boole` полезна, когда необходимо параметризировать процедуру так, что они может использовать одну из нескольких логических операций.

[Функция] **lognot** *integer*

Функция возвращает битовое логическое *отрицание* аргумента. Каждый бит результата является дополнение соответствующего исходного бита аргумента.

(logbitp j (lognot x)) ≡ (not (logbitp j x))

[Функция] **logtest** *integer1 integer2*

`logtest` является предикатом, который истинен, если любой бит определённый как 1 в *integer1* также является соответствующим битом 1 в *integer2*.

(logtest x y) ≡ (not (zerop (logand x y)))

[Функция] **logbitp** *index integer*

Предикат `logbitp` является истиной, если бит в позиции *index* в целом числе *integer* (то есть, его вес 2^{index}), является 1, иначе предикат ложен. Например:

(logbitp 2 6) is true
(logbitp 0 6) is false
(logbitp k n) ≡ (ldb-test (byte 1 k) n)

index должен быть неотрицательным целым числом.

[Функция] **ash** *integer count*

Если *count* положительное число, данная функция арифметически сдвигает число *integer* влево на количество бит *count*. Если *count* отрицательное число, функция сдвигает число *integer* вправо. Знак результата всегда такое же как и исходного числа *integer*.

Говоря математически, эта операция выполняет вычисления $floor(integer \cdot 2^{count})$.

Логически, функция перемещает все биты числа *integer* влево, добавляя нулевые биты в конец, или вправо отбрасывая биты в конце числа. (В этом контексте вопрос о том, что сдвигается налево, не относится к делу. Целые числа, рассматриваемые как строки битов, являются «полубесконечными», то есть концептуально расширяются бесконечно влево FIXME.) Например:

(logbitp j (ash n k)) ≡ (and (>= j k) (logbitp (- j k) n))

[Функция] **logcount** *integer*

Функция возвращает количество бит в числе *integer*. Если *integer* положительное число, тогда подсчитаны будут биты `1`. Если число *integer* отрицательно, то в два раза дополненной (two's-complement) бинарной форме будут подсчитаны биты `0`. FIXME Результатом всегда является неотрицательное целое число. Например:

(logcount 13) ⇒ 3 ;Бинарное представление ...0001101
(logcount -13) ⇒ 2 ;Бинарное представление ...1110011
(logcount 30) ⇒ 4 ;Бинарное представление ...0011110
(logcount -30) ⇒ 4 ;Бинарное представление ...1100010

Следующее тождество всегда верно:

(logcount x) ≡ (logcount (- (+ x 1)))
≡ (logcount (lognot x))

[Функция] **integer-length** *integer*

Данная функция выполняет следующие вычисления

$$ceiling(\log_2(\textbf{if } integer < 0 \textbf{ then } -integer \textbf{ else } integer + 1))$$

Эта функция полезна в двух случаях. Первое, если целое число *integer* не отрицательно, тогда его значение может быть представлено в беззнаковой бинарной форме в поле, ширина которого в битах не меньше чем `(integer-length` ***integer***`)`. Второе, независимо от знака числа *integer*, его значение может быть представлено как знаковая бинарная два раза дополненная (two's-complement) форма в поле, ширина которого в битах не меньше чем `(+ (integer-length` ***integer***`) 1)`. FIXME Например:

(integer-length 0) ⇒ 0
(integer-length 1) ⇒ 1
(integer-length 3) ⇒ 2
(integer-length 4) ⇒ 3
(integer-length 7) ⇒ 3
(integer-length -1) ⇒ 0

(integer-length -4) ⇒ 2
(integer-length -7) ⇒ 3
(integer-length -8) ⇒ 3

12.8 Функции для манипуляции с байтами

Common Lisp содержит несколько функций для работы с полями смежных битов произвольной длины, находящимися где-нибудь в целом числе. Такое множество смежных битов называется *байт*. В Common Lisp'е термин *байт* не означает некоторое определённое количество бит (как например восемь), а обозначает поле произвольной, определяемой пользователем длины.

Функции обработки байтом используют объекты, называющиеся *спецификаторы байта* для определения заданной позиции байта в целом числе. Представление спецификатора байта зависит от реализации. В частности оно может быть или не быть числом. Достаточно знать, что функция `byte` будет создавать такой спецификатор, и что функция для обработки байта будет принимать его. Функция `byte` принимает два целых числа указывающих на *позицию* и *размер* байта и возвращает его спецификатор. Такой спецификатор определяет байт, у которого ширина равна *размеру*, и биты которого имеют веса с $2^{position+size-1}$ по $2^{position}$.

[Функция] **byte** *size position*

`byte` принимает два целых числа, указывающих на размер и позицию байта и возвращает спецификатор, который может использоваться в функциях работы с байтами.

[Функция] **byte-size** *bytespec*
[Функция] **byte-position** *bytespec*

Принимая спецификатор байта, `byte-size` возвращает размер указанного байта, а `byte-position` — позицию этого байта. Например:

(byte-size (byte j k)) $\equiv j$
(byte-position (byte j k)) $\equiv k$

[Функция] **ldb** *bytespec integer*

Спецификатор байта *bytespec* указывает на байт, который будет извлечён из целого числа *integer*. Результат возвращается в качестве неотрицательного целого числа. Например:

(logbitp j (ldb (byte s p) n)) ≡ (and ($<$ j s) (logbitp ($+$ j p) n))

Имя функции `ldb` означает «load byte (загрузить байт)».

Если аргумент *integer* задан формой, которая может использоваться в `setf`, тогда `setf` может использоваться вместе с `ldb` для изменения байта в целом числе указанном в форме *integer*. Это действие выполняется с помощью операции `dpb`.

[Функция] **ldb-test** *bytespec integer*

`ldb-test` является предикатом, который истинен, если любой из битов, указанных в спецификаторе байта *bytespec*, в целом числе *integer* имеет значение 1. Другими словами, предикат истинен, если указанное поле не равно нулю.

(ldb-test *bytespec* n) ≡ (not (zerop (ldb *bytespec* n)))

[Функция] **mask-field** *bytespec integer*

Функция похожа а `ldb`. Однако результат содержит указанный байт целого числа *integer* в той же позиции, что указана в спецификаторе *bytespec*, а не в нулевой позиции, как делает `ldb`. Таким образом результат в указанном байте совпадает с целым числом *integer* , но в остальных местах имеет нулевые биты. Например:

(ldb *bs* (mask-field *bs* n)) ≡ (ldb *bs* n)

(logbitp j (mask-field (byte s p) n))
≡ (and ($>=$ j p) ($<$ j ($+$ p s)) (logbitp j n))

(mask-field *bs* n) ≡ (logand n (dpb -1 *bs* 0))

Если аргумент *integer* задан формой, которая может использоваться в `setf`, тогда `setf` может использоваться вместе с `mask-field` для изменения байта в целом числе указанном в форме *integer*. Это действие выполняется с помощью операции `deposit-field`.

[Функция] **dpb** *newbyte bytespec integer*

Данная функция возвращает число, которое похоже на *integer* за исключением битов, указанных спецификатором *bytespec*.. Пусть *s* будет размером, указанным в спецификаторе *bytespec*. Тогда целое число *newbyte* будет интерпретировано, будучи выровненным вправо, как если бы было результатом `ldb`. Например:

```
(logbitp j (dpb m (byte s p) n))
  ≡ (if (and (>= j p) (< j (+ p s)))
        (logbitp (- j p) m)
        (logbitp j n))
```

Имя функции `dpb` означает «deposit byte (сохранить байт)».

[Функция] **deposit-field** *newbyte bytespec integer*

Данная функция относится к `mask-field` так же, как `dpb` относится к `ldb`. Результатом является целое число, которое содержит биты *newbyte* в том месте, куда указывает спецификатор *bytespec*, и биты числа *integer* во всех остальных местах.

```
(logbitp j (deposit-field m (byte s p) n))
  ≡ (if (and (>= j p) (< j (+ p s)))
        (logbitp j m)
        (logbitp j n))
```

Заметка для реализации: If the *bytespec* is a constant, one may of course construct, at compile time, an equivalent mask *m*, for example by computing `(deposit-field -1 bytespec 0)`. Given this mask *m*, one may then compute

(deposit-field *newbyte bytespec integer*)

by computing

(logior (logand *newbyte* *m*) (logand *integer* (lognot *m*)))

where the result of `(lognot m)` can of course also be computed at compile time. However, the following expression may also be used and may require fewer temporary registers in some situations:

(logxor *integer* (logand *m* (logxor *integer* *newbyte*)))

A related, though possibly less useful, trick is that

```
(let ((z (logand (logxor x y) m)))
  (setq x (logxor z x))
  (setq y (logxor z y)))
```

interchanges those bits of `x` and `y` for which the mask `m` is `1`, and leaves alone those bits of `x` and `y` for which `m` is `0`.

12.9 Случайные числа

Функциональность Common Lisp'a для генерации псевдослучайных чисел была аккуратно проработана, чтобы быть переносимой. Тогда как две реализации могут создавать различные ряды псевдослучайных чисел, распределение значений должно быть независимо от таких характеристик компьютера как размер слова.

[Функция] **random** *number* **&optional** *state*

`(random n)` принимает положительное число n и возвращает число такого же типа между нулём (включительно) и n (невключительно). Число n может быть целым или с плавающей точкой. При этом используется приблизительно равномерное распределения. FIXME Если n целое число, каждый из возможных результатов встречается с (примерной) вероятностью $1/n$. (Слово «примерно» используется из-за

различий реализаций. На практике отклонение от однородности должно быть весьма мало.)

Аргумент *state* должен быть объектом типа `random-state`. По-умолчанию он равен значению переменной `*random-state*`. Объект используется для сохранения состояния генератора псевдослучайных чисел и изменяется как побочное действие от операции `random`.

Заметка для реализации: In general, even if `random` of zero arguments were defined as in MacLisp, it is not adequate to define `(random n)` for integral n to be simply `(mod (random) n)`; this fails to be uniformly distributed if n is larger than the largest number produced by `random`, or even if n merely approaches this number. This is another reason for omitting `random` of zero arguments in Common Lisp. Assuming that the underlying mechanism produces "random bits" (possibly in chunks such as fixnums), the best approach is to produce enough random bits to construct an integer k some number d of bits larger than `(integer-length n)` (see `integer-length`), and then compute `(mod k n)`. The quantity d should be at least 7, and preferably 10 or more.

To produce random floating-point numbers in the half-open range $[A, B)$, accepted practice (as determined by a look through the *Collected Algorithms from the ACM*, particularly algorithms 133, 266, 294, and 370) is to compute $X \cdot (B - A) + A$, where X is a floating-point number uniformly distributed over $[0.0, 1.0)$ and computed by calculating a random integer N in the range $[0, M)$ (typically by a multiplicative-congruential or linear-congruential method mod M) and then setting $X = N/M$. See also [27]. If one takes $M = 2^f$, where f is the length of the significand of a floating-point number (and it is in fact common to choose M to be a power of 2), then this method is equivalent to the following assembly-language-level procedure. Assume the representation has no hidden bit. Take a floating-point 0.5, and clobber its entire significand with random bits. Normalize the result if necessary.

For example, on the DEC PDP-10, assume that accumulator `T` is completely random (all 36 bits are random). Then the code sequence

```
LSH T,-9          ;Clear high 9 bits; low 27 are random
FSC T,128.        ;Install exponent and normalize
```

will produce in `T` a random floating-point number uniformly distributed over $[0.0, 1.0)$. (Instead of the `LSH` instruction, one could do

```
TLZ T,777000          ;That's 777000 octal
```

but if the 36 random bits came from a congruential random-number generator, the high-order bits tend to be "more random" than the low-order ones, and so the `LSH` would be better for uniform distribution. Ideally all the bits would be the result of high-quality randomness.)

With a hidden-bit representation, normalization is not a problem, but dealing with the hidden bit is. The method can be adapted as follows. Take a floating-point 1.0 and clobber the explicit significand bits with random bits; this produces a random floating-point number in the range $[1.0, 2.0)$. Then simply subtract 1.0. In effect, we let the hidden bit creep in and then subtract it away again.

For example, on the DEC VAX, assume that register `T` is completely random (but a little less random than on the PDP-10, as it has only 32 random bits). Then the code sequence

```
INSV #^X81,#7,#9,T      ;Install correct sign bit and exponent
SUBF #^F1.0,T           ;Subtract 1.0
```

will produce in `T` a random floating-point number uniformly distributed over $[0.0, 1.0)$. Again, if the low-order bits are not random enough, then the instruction

```
ROTL #7,T
```

should be performed first.

Implementors may wish to consult reference [41] for a discussion of some efficient methods of generating pseudo-random numbers.

[Переменная] ***random-state***

Данная переменная содержит объект типа `random-state`, который хранит состояние для генератора случайных чисел, которое по-умолчанию используется функцией `random`. Природа этой структуры данных зависит от реализации. Она может быть выведена на печать и прочитана обратно, но не обязательно это может быть сделано между реализациями. Вызов `random` изменяет этот объект. Связывание этой переменной с другим значением будет корректно сохраняться и восстанавливаться.

[Функция] **make-random-state &optional** *state*

Данная функция возвращает новый объект типа `random-state`, подходящий для использования в качестве значения переменной

`*random-state*`. Если *state* не указано или равно `nil`, тогда `make-random-state` возвращает *копию* текущего объекта состояния генератора псевдослучайных чисел (значение переменной `*random-state*`). Если *state* является объектом состояния, то возвращается копия данного объекта. Если *state* равен `t`, тогда новый возвращается новый объект состояния, который в некотором смысле «случайно» инициализирован (текущим временем например).

Обоснование: Common Lisp purposely provides no way to initialize a `random-state` object from a user-specified "seed." The reason for this is that the number of bits of state information in a `random-state` object may vary widely from one implementation to another, and there is no simple way to guarantee that any user-specified seed value will be "random enough." Instead, the initialization of `random-state` objects is left to the implementor in the case where the argument `t` is given to `make-random-state`.

To handle the common situation of executing the same program many times in a reproducible manner, where that program uses `random`, the following procedure may be used:

1. Evaluate `(make-random-state t)` to create a `random-state` object.

2. Write that object to a file, using `print`, for later use.

3. Whenever the program is to be run, first use `read` to create a copy of the `random-state` object from the printed representation in the file. Then use the `random-state` object newly created by the `read` operation to initialize the random-number generator for the program.

It is for the sake of this procedure for reproducible execution that implementations are required to provide a read/print syntax for objects of type `random-state`.

It is also possible to make copies of a `random-state` object directly without going through the print/read process, simply by using the `make-random-state` function to copy the object; this allows the same sequence of random numbers to be generated many times within a single program.

Заметка для реализации: A recommended way to implement the type `random-state` is effectively to use the machinery for `defstruct`. The usual structure syntax may then be used for printing `random-state` objects; one might look something like

```
#S(RANDOM-STATE DATA #(14 49 98436589 786345 8734658324 ...))
```

where the components are of course completely implementation-dependent.

[Функция] **random-state-p** *object*

`random-state-p` истинен, если аргумент является объектом состояния генератора случайных чисел, иначе — ложен.

(random-state-p x) ≡ (typep x 'random-state)

12.10 Параметры реализации

Значение констант, определённый в этом разделе, зависят от реализации. В некоторых случаях они могут быть полезны для параметризации кода.

[Константа] **most-positive-fixnum**
[Константа] **most-negative-fixnum**

Значением `most-positive-fixnum` является такое число, которое ближе всего к положительной бесконечности.

Значением `most-negative-fixnum` является такое число, которое ближе всего к отрицательной бесконечности.

X3J13 voted in January 1989 to specify that `fixnum` must be a supertype of the type `(signed-byte 16)`, and additionally that the value of `array-dimension-limit` must be a fixnum. This implies that the value of `most-negative-fixnum` must be less than or equal to -2^{15}, and the value of `most-positive-fixnum` must be greater than or equal to both $2^{15}-1$ and the value of `array-dimension-limit`.

[Константа] **most-positive-short-float**
[Константа] **least-positive-short-float**
[Константа] **least-negative-short-float**
[Константа] **most-negative-short-float**

Значением `most-positive-short-float` является такое значение числа короткого формата с плавающей точкой, которое ближе всего к (но не равно) положительной бесконечности.

Значением `least-positive-short-float` является такое значение числа короткого формата с плавающей точкой, которое ближе всего к (но не равно) нулю.

Значением `least-negative-short-float` является такое значение числа короткого формата с плавающей точкой, которое ближе всего к (но не равно) нулю слева. (Следует отметить, что если разработчики поддерживают минус ноль, то значение данной константы не может ему равняться.)

X3J13 voted in June 1989 to clarify that these definitions are to be taken quite literally. In implementations that support denormalized numbers, the values of `least-positive-short-float` and `least-negative-short-float` may be denormalized.

Значением `most-negative-short-float` является такое значение числа короткого формата с плавающей точкой, которое ближе всего к (но не равно) отрицательной бесконечности.

[Константа] **most-positive-single-float**
[Константа] **least-positive-single-float**
[Константа] **least-negative-single-float**
[Константа] **most-negative-single-float**
[Константа] **most-positive-double-float**
[Константа] **least-positive-double-float**
[Константа] **least-negative-double-float**
[Константа] **most-negative-double-float**
[Константа] **most-positive-long-float**
[Константа] **least-positive-long-float**
[Константа] **least-negative-long-float**
[Константа] **most-negative-long-float**

Данные константы аналогичны предыдущим, но указывают на другие форматы чисел с плавающей точкой.

[Константа] **least-positive-normalized-short-float**
[Константа] **least-negative-normalized-short-float**

Значением `least-positive-normalized-short-float` является такое положительное нормализованное число в коротком формате с плавающей запятой, которое ближе всего к (но не равно) нулю. В

реализации, которая не поддерживает денормализованные числа, может равняться тому же, что и `least-positive-short-float`.

Значением `least-positive-normalized-short-float` является такое отрицательное нормализованное число в коротком формате с плавающей запятой, которое ближе всего к (но не равно) нулю. (Следует отметить, что если реализация поддерживает минус ноль, то данная константа должна ему равняться.) В реализации, которая не поддерживает денормализованные числа, может равняться тому же, что и `least-negative-short-float`.

[Константа] **least-positive-normalized-single-float**
[Константа] **least-negative-normalized-single-float**
[Константа] **least-positive-normalized-double-float**
[Константа] **least-negative-normalized-double-float**
[Константа] **least-positive-normalized-long-float**
[Константа] **least-negative-normalized-long-float**

Данные константы аналогичны предыдущим, но указывают на другие форматы чисел с плавающей точкой.

[Константа] **short-float-epsilon**
[Константа] **single-float-epsilon**
[Константа] **double-float-epsilon**
[Константа] **long-float-epsilon**

Эти константы для каждого формата с плавающей точкой содержат наименьшее натуральное число с плавающей точкой e, такое что при вычислении выражение

(not (= (float 1 e) (+ (float 1 e) e)))

истинно.

[Константа] **short-float-negative-epsilon**
[Константа] **single-float-negative-epsilon**
[Константа] **double-float-negative-epsilon**
[Константа] **long-float-negative-epsilon**

Эти константы для каждого формата с плавающей точкой содержат

наименьшее натуральное число с плавающей точкой e, такое что при вычислении выражение

(not (= (float 1 e) (- (float 1 e) e)))

истинно.

Глава 13

Строковые символы

Common Lisp содержит тип данный строковые символы. Объекты данного типа представляют печатаемые символы, как например буквы.

Строковые символы в Common Lisp'е не совсем объекты. Нельзя положится на то, что `eq` будет работать с ними правильно. В частности, возможно что выражение

(let ((x z) (y z)) (eq x y))

может быть ложным, а не истины, если значение `z` является строковым символом.

Обоснование: Необязательное равенство символов с помощью `eq` позволяет разработчикам реализации сделать оптимизации на тех архитектурах, где это удобно. Такое же правило сделано и для чисел, смотрите главу 12.

Для сравнения двух объектов, один из которых может быть символом, необходимо использовать предикат `eql`.

13.1 Свойство строковых символов

[Константа] **char-code-limit**

Значением `char-code-limit` является неотрицательное целое число, которое отображает наибольшее из возможных значений функции

Таблица 13.1: Стандартные метки символов, символы и описания

	SM05@собака	SD13‘обратная кавычка
SP02!восклицательный знак	LA02Aпрописная A	LA01aмаленькая a
SP04"двойная кавычка	LB02Bпрописная B	LB01bмаленькая b
SM01#диез, решётка	LC02Cпрописная C	LC01cмаленькая c
SC03$знак доллара	LD02Dпрописная D	LD01dмаленькая d
SM02%знак процента	LE02Eпрописная E	LE01eмаленькая e
SM03&амперсанд	LF02Fпрописная F	LF01fмаленькая f
SP05’апостроф	LG02Gпрописная G	LG01gмаленькая g
SP06(левая круглая скобка	LH02Hпрописная H	LH01hмаленькая h
SP07)права круглая скобка	LI02Iпрописная I	LI01iмаленькая i
SM04*звёздочка	LJ02Jпрописная J	LJ01jмаленькая j
SA01+знак плюс	LK02Kпрописная K	LK01kмаленькая k
SP08,запятая	LL02Lпрописная L	LL01lмаленькая l
SP10-дефис или знак минус	LM02Mпрописная M	LM01mмаленькая m
SP11.точка	LN02Nпрописная N	LN01nмаленькая n
SP12/слеш	LO02Oпрописная O	LO01oмаленькая o
ND100цифра 0	LP02Pпрописная P	LP01pмаленькая p
ND011цифра 1	LQ02Qпрописная Q	LQ01qмаленькая q
ND022цифра 2	LR02Rпрописная R	LR01rмаленькая r
ND033цифра 3	LS02Sпрописная S	LS01sмаленькая s
ND044цифра 4	LT02Tпрописная T	LT01tмаленькая t
ND055цифра 5	LU02Uпрописная U	LU01uмаленькая u
ND066цифра 6	LV02Vпрописная V	LV01vмаленькая v
ND077цифра 7	LW02Wпрописная W	LW01wмаленькая w
ND088цифра 8	LX02Xпрописная X	LX01xмаленькая x
ND099цифра 9	LY02Yпрописная Y	LY01yмаленькая y
SP13:двоеточие	LZ02Zпрописная Z	LZ01zмаленькая z
SP14;точка с запятой	SM06[левая квадратная скобка	SM11{левая фигурная скобка
SA03<знак меньше чем	SM07\обратный слеш	SM13|вертикальная черта
SA04=знак равенства	SM08]правая квадратная скобка	SM14}правая фигурная скобк
SA05>знак больше чем	SD15^крыша	SD19~тильда
SP15?вопросительный знак	SP09_знак подчёркивания	

Символы в этой таблице, а также пробел и символы новой строки составляют стандартный набор символов для Common Lisp’a (тип `standard-char`). Метки символов и описания символов взяты из ISO 6937/2. Первый символ на метке классифицирует символ как Latin, Numeric или Special.

`char-code` невключительно. То есть значения, возвращаемые `char-code` неотрицательны и строго меньше чем значение `char-code-limit`.

Common Lisp не гарантирует, что все целые числа между нулём и `char-code-limit` являются корректными кодами для символов.

13.2 Предикаты для строковых символов

Предикат `characterp` используется для определения, является ли Lisp'овый объект строковым символом.

[Функция] **standard-char-p** *char*

Аргумент *char* должен быть строковым символом. `standard-char-p` истинен, если аргумент является «стандартным строковым символом», то есть объект принадлежит типу `standard-char`.

Следует отметить, что любой символ с ненулевым битом или свойством шрифта не является стандартным.

[Функция] **graphic-char-p** *char*

Аргумент *char* должен быть строковым символом. `graphic-char-p` истинен, если аргумент является «графическим» (выводимым) символом, или ложен, если аргумент является «неграфическим» (форматирующим или управляющим) символом. Графические символы имеют стандартное текстовое представление в качестве одного знака, такого как например `A` или `*` или `=`. По соглашению, символы пробела рассматриваются как графические. Все стандартные символы за исключением `#\Newline` являются графическими. Не совсем стандартные символы `#\Backspace`, `#\Tab`, `#\Rubout`, `#\Linefeed`, `#\Return` и `#\Page` графическими не являются.

[Функция] **alpha-char-p** *char*

Аргумент *char* должен быть строковым символом. `alpha-char-p` истинен, если аргумент являются алфавитным символом, иначе предикат ложен.

Если символ является алфавитным, тогда он является графическим.

Из стандартных символов (как определено с помощью `standard-char-p`), буквы с `A` по `Z` и с `a` по `z` являются алфавитными.

[Функция] **upper-case-p** *char*
[Функция] **lower-case-p** *char*
[Функция] **both-case-p** *char*

Аргумент *char* должен быть строковым символом.

`upper-case-p` истинен, если аргумент является символом в верхнем регистре, иначе ложен.

`lower-case-p` истинен, если аргумент является символом в нижнем регистре, иначе ложен.

`both-case-p` истинен, если аргумент является символом в верхнем регистре, и для этого символа существует аналогичный в нижнем регистре (это может быть установлено с помощью `char-downcase`), или если аргумент является символом в нижнем регистре, и для этого символа существует аналогичный в верхнем регистре (это может быть установлено с помощью `char-upcase`).

Из стандартных символов (как определено с помощью `standard-char-p`), буквы с `A` по `Z` имеют верхний регистр и буквы с `a` по `z` нижний.

[Функция] **digit-char-p** *char* **&optional** (*radix* 10)

Аргумент *char* должен быть строковым символом, и *radix* неотрицательным целым числом. Если *char* не является цифрой для указанной в *radix* системы счисления, тогда `digit-char-p` ложен, иначе предикат возвращает значение данного символа в этой системе счисления.

Цифры принадлежат графическим символам.

Из стандартных символов (как определено с помощью `standard-char-p`), символы с `0` по `9`, с `A` по `Z` и с `a` по `z` являются цифровыми. Веса с `0` по `9` совпадают с числами с 0 по 9, и с `A` по `Z` (а также с `a` по `z`) совпадают с числами с 10 по 35. `digit-char-p` возвращает вес одной их этих цифр тогда и только тогда, когда их вес строго меньше чем *radix*. Таким образом, например, цифры для шестнадцатеричной системы счисления будут такими

0 1 2 3 4 5 6 7 8 9 A B C D E F

Пример использования `digit-char-p`:

```
(defun convert-string-to-integer (str &optional (radix 10))
 "Принимает строку и опционально систему счисления, возвращает целое число."
 (do ((j 0 (+ j 1))
      (n 0 (+ (* n radix)
              (or (digit-char-p (char str j) radix)
                  (error "Bad radix-~D digit: ~C"
                         radix
                         (char str j))))))
     ((= j (length str)) n)))
```

[Функция] **alphanumericp** *char*

Аргумент *char* должен быть строковым символом. Предикат `alphanumericp` истинен, если *char* является буквой или цифрой. Определение:

```
(alphanumericp x)
  ≡ (or (alpha-char-p x) (not (null (digit-char-p x))))
```

Таким образом алфавитно-цифровой символ обязательно является графическим (в соответствии с предикатом `graphic-char-p`).

Из стандартных символов (в соответствие с предикатом `standard-char-p`), символы с `0` по `9`, с `A` по `Z`, с `a` по `z` являются алфавитно-цифровыми.

[Функция] **char=** *character* **&rest** *more-characters*
[Функция] **char/=** *character* **&rest** *more-characters*
[Функция] **char<** *character* **&rest** *more-characters*
[Функция] **char>** *character* **&rest** *more-characters*
[Функция] **char<=** *character* **&rest** *more-characters*
[Функция] **char>=** *character* **&rest** *more-characters*

Все аргументы должны быть строковыми символами. Данный функции сравнивают символы методом зависящим от реализации.

Порядок расположения строковых символов гарантированно удовлетворяет следующим правилам:

- Стандартные алфавитно-цифровые символы подчиняются следующему порядку:

  A<B<C<D<E<F<G<H<I<J<K<L<M<N<O<P<Q<R<S<T<U<V<W<X<Y<Z
  a<b<c<d<e<f<g<h<i<j<k<l<m<n<o<p<q<r<s<t<u<v<w<x<y<z
  0<1<2<3<4<5<6<7<8<9
  одно из двух 9<A *или* Z<0
  одно из двух 9<a *или* z<0

Порядок следования символов необязательно совпадает с порядком следования их кодов, полученных из функции `char-int`.

Порядок следование символов не является неразрывным. Таким образом выражение `(char<= #\a x #\z)` нельзя использовать для проверки является ли `x` символом в нижнем регистре. Для этого предназначен предикат `lower-case-p`.

(char= #\d #\d) истина.
(char/= #\d #\d) ложь.
(char= #\d #\x) ложь.
(char/= #\d #\x) истина.
(char= #\d #\D) ложь.
(char/= #\d #\D) истина.
(char= #\d #\d #\d #\d) истина.
(char/= #\d #\d #\d #\d) ложь.
(char= #\d #\d #\x #\d) ложь.
(char/= #\d #\d #\x #\d) ложь.
(char= #\d #\y #\x #\c) ложь.
(char/= #\d #\y #\x #\c) истина.
(char= #\d #\c #\d) ложь.
(char/= #\d #\c #\d) ложь.
(char< #\d #\x) истина.
(char<= #\d #\x) истина.
(char< #\d #\d) ложь.
(char<= #\d #\d) истина.
(char< #\a #\e #\y #\z) истина.
(char<= #\a #\e #\y #\z) истина.

(char< #\a #\e #\e #\y) ложь.
(char<= #\a #\e #\e #\y) истина.
(char> #\e #\d) истина.
(char>= #\e #\d) истина.
(char> #\d #\c #\b #\a) истина.
(char>= #\d #\c #\b #\a) истина.
(char> #\d #\d #\c #\a) ложь.
(char>= #\d #\d #\c #\a) истина.
(char> #\e #\d #\b #\c #\a) ложь.
(char>= #\e #\d #\b #\c #\a) ложь.
(char> #\z #\A) может быть истиной или ложью.
(char> #\Z #\a) может быть истиной или ложью.

Если и `(char= c1 c2)` является истиной, то `(eq c1 c2)` истиной может и не являться. `eq` сравнивает строковые символы не как символы, а как объекты с различием в свойствах, которое зависит от конкретной реализации. (Конечно, если `(eq c1 c2)` истинно, то `(char= c1 c2)` также будет истинно.) Однако, `eql` и `equal` сравнивают строковые символы также как и `char=`.

[Функция] **char-equal** *character* **&rest** *more-characters*
[Функция] **char-not-equal** *character* **&rest** *more-characters*
[Функция] **char-lessp** *character* **&rest** *more-characters*
[Функция] **char-greaterp** *character* **&rest** *more-characters*
[Функция] **char-not-greaterp** *character* **&rest** *more-characters*
[Функция] **char-not-lessp** *character* **&rest** *more-characters*

Для стандартных символов порядок между ними такой, что выполняются равенства `A=a`, `B=b` и так до `Z=z`, а также выполняется одно из двух неравенств `9<A` или `Z<0`.

(char-equal #\A #\a) истина.
(char= #\A #\a) ложь.
(char-equal #\A #\Control-A) истина.

13.3 Код символа

[Функция] **char-code** *char*

Аргумента *char* должен быть строковым объектом. `char-code` возвращает код символа, а именно неотрицательное целое число, меньшее чем значение переменной `char-code-limit`.

Однако помните, не все целые числа на это промежутке могут быть корректными отображениями символов.

[Функция] **code-char** *code*

Возвращает строковый символ для заданного кода. Если для этого кода символа не существует, тогда возвращается `nil`. Например:

(char= (code-char (char-code c)) c)

13.4 Преобразование строковых символов

Данные функции выполняют различные преобразования символов, включая изменение регистра.

[Функция] **character** *object*

Функция `character` если возможно возвращает преобразованный в символ аргумент *object*. Смотрите `coerce`.

(character x) ≡ (coerce x 'character)

[Функция] **char-upcase** *char*
[Функция] **char-downcase** *char*

Аргумент *char* должен быть строковым символом. `char-upcase` пытается возвести символ в верхний регистр. `char-downcase` пытается возвести символ в нижний регистр.

[Функция] **digit-char** *weight* **&optional** (*radix* 10)

Все аргументы должны быть целыми числами. `digit-char` устанавливает может ли быть создан символ, у которого код *code* такой, что итоговый символ имеет вес *weight*, когда рассматривается как цифра системы счисления *radix* (смотрите предикат `digit-char-p`). В случае успеха возвращается этот символ, иначе `nil`.

`digit-char` не может вернуть `nil`, если *radix* находится между 2 и 36 включительно и *weight* имеет неотрицательное значение и меньше чем *radix*.

Если для результата подходят несколько символов, выбор лежит на плечах реализации. Но символы в верхнем регистре предпочтительнее символов в нижем. Например:

(digit-char 7) ⇒ #\7
(digit-char 12) ⇒ `nil`
(digit-char 12 16) ⇒ #\C ;not #\c
(digit-char 6 2) ⇒ `nil`
(digit-char 1 2) ⇒ #\1

[Функция] **char-int** *char*

Аргумент *char* должен быть строковым символом. `char-int` возвращает неотрицательный целый числовой код символа.

Следует отметить, что

(char= c1 c2) ≡ (= (char-int c1) (char-int c2))

для любых символов `c1` и `c2`
Данная функция создана в основном для хеширования символов.

[Функция] **char-name** *char*

Аргумент *char* должен быть строковым символом. Если строковый символ имеет имя, то результатом будет это имя (в виде строки), иначе результат будет `nil`. Имена есть у всех неграфических (не удовлетворяющих предикату `graphic-char-p`).

Стандартные символы перевода строки и проблема имеют имена `Newline` и `Space`. Полустандартные символы имеют имена `Tab`, `Page`, `Rubout`, `Linefeed`, `Return` и `Backspace`.

Символы, у которых есть имена, могут быть заданы с помощью `#\` и последующего имени. (Смотрите раздел 22.1.4.) Имя может быть записано в любом регистре, но основной стиль предполагает запись просто с большой буквы `#\Space`.

[Функция] **name-char** *name*

Аргумент *name* должен быть объектом, который можно превратить в строку, например, с помощью функции `string`. Если имя совпадает с именем некоторого строкового символа (проверка осуществляется с помощью `string-equal`), тогда будет возвращён этот символ, иначе возвращается `nil`.

Глава 14

Последовательности

Тип `sequence` объединяет списки и вектора (одномерные массивы). Хотя это две различные структуры данных с различными структурными свойствами, приводящими к алгоритмически различному использованию, они имеют общее свойство: каждая хранит упорядоченное множество элементов.

Некоторые операции полезны и для списков и для массивов, потому что они взаимодействуют с упорядоченными множествами элементов. Можно запросить количество элементов, изменить порядок элементов на противоположный, извлечь подмножество (подпоследовательность) и так далее. Для таких целей Common Lisp предлагает ряд общих для всех типов последовательностей функций.

elt	reverse	map	remove
length	nreverse	some	remove-duplicates
subseq	concatenate	every	delete
copy-seq	position	notany	delete-duplicates
fill	find	notevery	substitute
replace	sort	reduce	nsubstitute
count	merge	search	mismatch

Некоторые из этих операций имеют более одной версии. Такие версии отличаются суффиксом (или префиксом) от имени базовой функции. Кроме того, многие операции принимают один или более необязательных именованных аргументов, которые могут изменить поведение операции.

Если операция требует проверки элементов последовательности на совпадение с некоторым условием, тогда это условие может быть указано

одним из двух способов. Основная операция принимает объект и сравнивает с ним каждый элемент последовательности на равенство `eql`. (Операция сравнения может быть задана в именованном параметре `:test` или `:test-not`. Использование обоих параметров одновременно является ошибкой.) Другие варианты операции образуются с добавлением префиксов `-if` и `-if-not`. Эти операции, в отличие от базовой, принимают не объект, а предикат с одним аргументом. В этом случае проверяется истинность или ложность предиката для каждого элемента последовательности. В качестве примера,

(remove *item sequence*)

возвращает копию последовательности *sequence*, в которой удалены все элементы равные `eql` объекту *emph*.

(remove *item sequence* `:test` #'equal)

возвращает копию последовательности *sequence*, в которой удалены все элементы равные `equal` объекту *emph*.

(remove-if #'numberp *sequence*)

возвращает копию последовательности *sequence*, в которой удалены все числа.

Если операция любым методом проверяет элементы последовательности, то если именованный параметр `:key` не равняется `nil`, то он должен быть функцией одного аргумента, которая будет извлекать из элемента необходимую для проверки часть. Например,

(find *item sequence* `:test` #'eq `:key` #'car)

Это выражение ищет первый элемент последовательности *sequence*, *car* элемент которого равен `eq` объекту *item*.

X3J13 voted in June 1988 to allow the `:key` function to be only of type `symbol` or `function`; a lambda-expression is no longer acceptable as a functional argument. One must use the `function` special operator or the abbreviation `#'` before a lambda-expression that appears as an explicit argument form.

Для некоторых операций было бы удобно указать направление обработки последовательности. В этом случае базовые операции обычно обрабатывают последовательность в прямом направлении. Обратное направление обработки указывается с помощью не-`nil` значения для именованного параметра `:from-end`. (Порядок обработки указываемый в `:from-end` чисто концептуальный. В зависимости от обрабатываемого объекта и реализации, действительный порядок обработки может отличаться. Поэтому пользовательские функции *test* не должны иметь побочных эффектов.)

Много операций позволяют указать подпоследовательность для обработки. Такие операции имеют именованные параметры `:start` и `:end`. Эти аргумент должны быть индексами внутри последовательности, и $start \leq end$. Ситуация $start > end$ является ошибкой. Эти параметры указывают подпоследовательность начиная с позиции *start* включительно и заканчивая позицией *end* невключительно. Таким образом длина подпоследовательности равна $end - start$. Если параметр *start* опущен, используется значение по-умолчанию ноль. Если параметр *end* опущен, используется значение по-умолчанию длина последовательности. В большинстве случаев, указание подпоследовательности допускается исключительно ради эффективности. Вместо этого можно было бы просто вызвать `subseq`. Однако, следует отметить, что операция, которая вычисляет индексы для подпоследовательности, возвращает индексы исходной последовательности, а не подпоследовательности:

(position #\b "foobar" `:start` 2 `:end` 5) ⇒ 3
(position #\b (subseq "foobar" 2 5)) ⇒ 1

Если в операции участвует две последовательности, тогда ключевые параметры `:start1`, `:end1`, `start2` и `:end2` используются для указания отдельных подпоследовательностей для каждой последовательности.

X3J13 voted in June 1988 (and further clarification was voted in January 1989) to specify that these rules apply not only to all built-in functions that have keyword parameters named `:start`, `:start1`, `:start2`, `:end`, `:end1`, or `:end2` but also to functions such as `subseq` that take required or optional parameters that are documented as being named *start* or *end*.

- A “start” argument must always be a non-negative integer and defaults to zero if not supplied; it is not permissible to pass `nil` as a “start” argument.

- An “end” argument must be either a non-negative integer or `nil` (which indicates the end of the sequence) and defaults to `nil` if not supplied; therefore supplying `nil` is equivalent to not supplying such an argument.

- If the “end” argument is an integer, it must be no greater than the active length of the corresponding sequence (as returned by the function `length`).

- The default value for the “end” argument is the active length of the corresponding sequence.

- The “start” value (after defaulting, if necessary) must not be greater than the corresponding “end” value (after defaulting, if necessary).

This may be summarized as follows. Let x be the sequence within which indices are to be considered. Let s be the “start” argument for that sequence of any standard function, whether explicitly specified or defaulted, through omission, to zero. Let e be the “end” argument for that sequence of any standard function, whether explicitly specified or defaulted, through omission or an explicitly passed `nil` value, to the active length of x, as returned by `length`. Then it is an error if the test `(<= 0 s e (length x))` is not true.

Для некоторых функций, в частности `remove` и `delete`, ключевой параметр `:count` используется для указания, как много подходящих элементов должны обрабатываться. Если он равен `nil` или не указан, обрабатываются все подходящие элементы.

В следующих описаниях функций, элемент x последовательности «удовлетворяет условию», если любое из следующих выражений верно:

- Была вызвана базовая функция, функция *testfn* заданная в параметре `:test`, и выражение `(funcall` *`testfn item (keyfn x)`*`)` истинно.

- Была вызвана базовая функция, функция *testfn* заданная в параметре `:test-not`, и выражение `(funcall` *`testfn item (keyfn x)`*`)` ложно.

- Была вызвана функция `-if`, и выражение (`funcall` *`predicate`* (*`keyfn x`*)) истинно.

- Была вызвана функция `-if-not`, и выражение (`funcall` *`predicate`* (*`keyfn x`*)) ложно

В каждом случае, функция *keyfn* является значением параметра `:key` (по-умолчанию функцией эквивалентности). Для примера, смотрите `remove`.

В следующих описаниях функций два элемента x и y, взятых из последовательности, «эквивалентны», если одно из следующих выражений верно:

- Функция *testfn* указана в параметре `:test`, и выражение (`funcall` *`testfn`* (*`keyfn x`*) (*`keyfn y`*)) истинно.

- Функция *testfn* указана в параметре `:test-not`, и выражение (`funcall` *`testfn`* (*`keyfn x`*) (*`keyfn y`*)) ложно.

Для примера, смотрите `search`.

X3J13 voted in June 1988 to allow the *testfn* or `predicate` to be only of type `symbol` or `function`; a lambda-expression is no longer acceptable as a functional argument. One must use the `function` special operator or the abbreviation `#'` before a lambda-expression that appears as an explicit argument form.

Вы можете рассчитывать на порядок передачи аргументов в функцию *testfn*. Это позволяет использовать некоммутативную функцию проверки в стиле предиката. Порядок аргументов в функцию *testfn* соответствует порядку, в котором эти аргументы (или последовательности их содержащие) были переданы в рассматриваемую функцию. Если рассматриваемая функция передаёт два элемента из одной последовательности в функцию *testfn*, то аргументы передаются в том же порядке, в котором были в последовательности.

Если функция должна создать и вернуть новый вектор, то она всегда возвращает *простой* вектор (смотрите раздел 2.5). Таким же образом, любые создаваемые строки будут простыми строками.

[Function] **complement** *fn*

Returns a function whose value is the same as that of `not` applied to the

result of applying the function *fn* to the same arguments. One could define `complement` as follows:

Данная функция возвращает другую функцию, значение которой является *отрицанием* функции *fn*. Функцию `complement` можно было определить так:

```
(defun complement (fn)
  #'(lambda (&rest arguments)
      (not (apply fn arguments))))
```

One intended use of `complement` is to supplant the use of `:test-not` arguments and `-if-not` functions.

Можно использовать функцию `complement` для эмуляции `-if-not` функций или аргумента `:test-not`.

```
(remove-if-not #'virtuous senators) ≡
  (remove-if (complement #'virtuous) senators)

(remove-duplicates telephone-book
                   :test-not #'mismatch) ≡
  (remove-duplicates telephone-book
                     :test (complement #'mismatch))
```

14.1 Простые функции для последовательностей

Большинство следующих функций выполняют простые операции над одиночными последовательностями. `make-sequence` создаёт новую последовательность.

[Функция] **elt** *sequence index*

Функция возвращает элемент последовательности *sequence*, указанный индексом *index*. Индекс должен быть неотрицательным

целым числом меньшим чем длина последовательности *sequence*. Длина последовательности, в свою очередь, вычисляется функцией `length`. Первый элемент последовательности имеет индекс `0`.

(Следует отметить, что `elt` учитывает указатель заполнения для векторов, его имеющих. Для доступа ко всем элементам таких векторов используется функция для массивов `aref`.)

Для изменения элемента последовательности может использоваться функция `setf` в связке с `elt`.

[Функция] **subseq** *sequence start* **&optional** *end*

Данная функция возвращает подпоследовательность последовательности *sequence* начиная с позиции *start* и заканчивая позицией *end*. `subseq` *всегда* создаёт новую последовательность для результата. Возвращённая подпоследовательность всегда имеет тот же тип, что и исходная последовательность.

Для изменения подпоследовательности элементов можно использовать `setf` в связке с `subseq`. Смотрите также `replace`.

[Функция] **copy-seq** *sequence*

Функция создаёт копию аргумента *sequence*. Результат равен `equalp` аргументу. Однако, результат не не равен `eq` аргументу.

(copy-seq x) $\equiv$ (subseq x 0)

но имя `copy-seq` лучше передаёт смысл операции.

[Функция] **length** *sequence*

Функция возвращает количество элементов последовательности *sequence*. Результат является неотрицательным целым числом. Если последовательность является вектором с указателем заполнения, возвращается «активная длина», то есть длина заданная указателем заполнения (смотрите раздел 17.5).

[Функция] **reverse** *sequence*

Результатом является новая последовательность такого же типа, что и последовательность *sequence*. Элементы итоговой

последовательности размещаются в обратном порядке. Аргумент при этом не модифицируется.

[Функция] **nreverse** *sequence*

Результатом является последовательность, содержащая те же элементы, что и последовательность *sequence*, но в обратном порядке. Аргумент может быть уничтожен и использован для возврата результата. Результат может быть или не быть равен `eq` аргументу. Поэтому обычно используют явное присваивание `(setq x (nreverse x))`, так как просто `(nreverse x)` не гарантирует возврат преобразованной последовательности в `x`.

X3J13 voted in March 1989 to clarify the permissible side effects of certain operations. When the *sequence* is a list, `nreverse` is permitted to perform a `setf` on any part, *car* or *cdr*, of the top-level list structure of that list. When the *sequence* is an array, `nreverse` is permitted to re-order the elements of the given array in order to produce the resulting array.

[Функция] **make-sequence** *type size* **&key** *:initial-element*

Данная функция возвращает последовательность типа *type* и длинной *size*, каждый из элементов которой содержит значение аргумента `:initial-element`. Если указан аргумент `:initial-element`, он должен принадлежать типу элемента последовательности *type*. Например:

```
(make-sequence '(vector double-float)
               100
               :initial-element 1d0)
```

Если аргумент `:initial-element` не указан, тогда последовательность инициализируется алгоритмом реализации.

X3J13 voted in January 1989 to clarify that the *type* argument must be a type specifier, and the *size* argument must be a non-negative integer less than the value of `array-dimension-limit`.

X3J13 voted in June 1989 to specify that `make-sequence` should signal an error if the sequence *type* specifies the number of elements and the *size* argument is different.

X3J13 voted in March 1989 to specify that if *type* is `string`, the result is the same as if `make-string` had been called with the same *size* and `:initial-element` arguments.

14.2 Объединение, отображение и приведение последовательностей

Функции в этом разделе могут обрабатывать произвольное количество последовательностей за исключением функции `reduce`, которая была включена сюда из-за концептуальной связи с функциями отображения.

[Функция] **concatenate** *result-type* **&rest** *sequences*

Результатом является новая последовательность, которая содержит по порядку все элементы указанных последовательностей. Результат не содержит связей с аргументами (в этом `concatenate` отличается от `append`). Тип результата указывается в аргументе *result-type*, который должен быть подтипом `sequence`, как для функции `coerce`. Необходимо, чтобы элементы исходных последовательностей принадлежали указанному типу *result-type*.

Если указана только одна последовательность, и она имеет тот же тип, что указан в *result-type*, `concatenate` всё равно копирует аргумент и возвращает новую последовательность. Если вам не требуется копия, а просто необходимо преобразовать тип последовательности, лучше использовать `coerce`.

X3J13 voted in June 1989 to specify that `concatenate` should signal an error if the sequence type specifies the number of elements and the sum of the argument lengths is different.

[Функция] **map** *result-type function sequence* **&rest** *more-sequences*

Функция *function* должна принимать только аргументов, сколько последовательностей было передано в `map`. Результат функции `map` — последовательность, элементы которой являются результатами применения функции *function* к соответствующим элементам исходных последовательностей. Длина итоговой последовательности равна длине самой короткой исходной.

Если функция *function* имеет побочные эффекты, она может рассчитывать на то, что сначала будет вызвана со всеми элементами в 0-ой позиции, затем со всеми в 1-ой и так далее.

Тип итоговой последовательности указывается в аргументе *result-type* (и должен быть подтипом `sequence`). Кроме того, для типа можно указать `nil`, и это означает, что результата быть не должно. В таком случае функция *function* вызывается только для побочных эффектов, и `map` возвращает `nil`. В этом случае `map` похожа на `mapc`.

X3J13 voted in June 1989 to specify that `map` should signal an error if the sequence type specifies the number of elements and the minimum of the argument lengths is different.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

Например:

```
(map 'list #'- '(1 2 3 4)) ⇒ (-1 -2 -3 -4)
(map 'string
     #'(lambda (x) (if (oddp x) #\1 #\0))
     '(1 2 3 4))
  ⇒ "1010"
```

[Функция] **map-into** *result-sequence function* **&rest** *sequences*

Функция `map-into` вычисляет последовательность с помощью применения функции *function* к соответствующим элементам исходных последовательностей и помещает результат в последовательность *result-sequence*. Функция возвращается *result-sequence*.

Аргументы *result-sequence* и все *sequences* должны быть списками или векторами (одномерными массивами). Функция *function* должна принимать столько аргументов, сколько было указано исходных последовательностей. Если *result-sequence* и другие аргументы *sequences* не одинаковой длины, цикл закончится на самой короткой последовательности. Если *result-sequence* является вектором с указателем заполнения, то этот указатель во время цикла игнорируется, а после завершения устанавливается на то количество элементов, которые были получены от функции *function*.

Если функция *function* имеет побочные эффекты, она может рассчитывать на то, что сначала будет вызвана со всеми элементами в 0-ой позиции, затем со всеми в 1-ой и так далее.

Функция `map-into` отличается от `map` тем, что модифицирует уже имеющуюся последовательность, а не создаёт новую. Кроме того, `map-into` может быть вызвана только с двумя аргументами (*result-sequence* и *function*), тогда как `map` требует как минимум три параметра.

Если *result-sequence* является `nil`, `map-into` немедленно возвращает `nil`, потому что длина `nil` последовательности равняется нулю.

[Функция] **some** *predicate sequence* **&rest** *more-sequences*
[Функция] **every** *predicate sequence* **&rest** *more-sequences*
[Функция] **notany** *predicate sequence* **&rest** *more-sequences*
[Функция] **notevery** *predicate sequence* **&rest** *more-sequences*

Всё это предикаты. Функция *predicate* должна принимать столько аргументов, сколько было указано последовательностей. Функция *predicate* сначала применяется к элементам `0`-ой позиции, затем, возможно, к элементам `1`-ой позиции, и так далее, до тех пор пока не сработает условие завершения цикла или не закончится одна из последовательностей *sequences*.

Если *predicate* имеет побочные эффекты, он может рассчитывать на то, что сначала будет вызвана со всеми элементами в `0`-ой позиции, затем со всеми в `1`-ой и так далее.

`some` завершается, как только применение *predicate* вернёт не-`nil` значение. Тогда `some` возвращает значение, на котором произошла остановка. Если цикл достиг конца последовательности и ни одно применение *predicate* не вернуло не-`nil`, тогда возвращается значение `nil`. Таким образом, можно сказать, что предикат `some` истинен, если *какой-либо (some)* вызов *predicate* истинен.

`every` возвращает `nil`, как только применение *predicate* возвращает `nil`. Если цикл достиг конца последовательности, `every` возвращает не-`nil`. Таким образом, можно сказать, что предикат `every` истинен, если *каждый (every)* вызов *predicate* истинен.

`notany` возвращает `nil`, как только применение *predicate* возвращает не-`nil`. Если цикл достиг конца последовательности, `notany` возвращает не-`nil`. Таким образом, можно сказать, что предикат `notany` истинен, если *ни какой (notany)* вызов *predicate* не истинен.

`notevery` возвращает не-`nil`, как только применение *predicate* возвращает `nil`. Если цикл достиг конца последовательности, `notevery`

возвращает `nil`. Таким образом, можно сказать, что предикат `notevery` истинен, если *ни каждый (notany)* вызов *predicate* истинен.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

[Функция] **reduce** *function sequence* **&key** *:from-end :start :end :initial-value*

Функция `reduce` объединяет все элементы последовательности использую заданную (бинарную binary) операцию. Например `+` может суммировать все элементы последовательности.

Указанная подпоследовательность последовательности *sequence* объединяется или «редуцируется» с помощью функции *function*, которая должна принимать два аргумента. Приведение (редуцирование, объединение) является левоассоциативным, если только `:from-end` не содержит истину, в последнем случае операция становиться правоассоциативной. По-умолчанию `:from-end` равно `nil`. Если задан аргумент `:initial-value`, то он логически помещается перед подпоследовательностью (или после в случае истинности `:from-end`) и включается в операцию редуцирования.

Если указанная подпоследовательность содержит только один элемент и параметр `:initial-value` не задан, тогда этот элемент возвращается, и функция *function* ни разу не вызывается. Если заданная подпоследовательность пуста, и задан параметр `:initial-value`, тогда возвращается `:initial-value`, и функция *function* не вызывается.

Если заданная подпоследовательность пуста, и параметр `:initial-value` не задан, тогда функция *function* вызывается без аргументов, и `reduce` возвращает то, что вернула эта функция. Это единственное исключение из правила о том, что функция *function* вызывается с двумя аргументами.

```
(reduce #'+ '(1 2 3 4)) ⇒ 10
(reduce #'- '(1 2 3 4)) ≡ (- (- (- 1 2) 3) 4) ⇒ -8
(reduce #'- '(1 2 3 4) :from-end t)     ;Альтернативная сумма
  ≡ (- 1 (- 2 (- 3 4))) ⇒ -2
(reduce #'+ '()) ⇒ 0
(reduce #'+ '(3)) ⇒ 3
(reduce #'+ '(foo)) ⇒ foo
```

```
(reduce #'list '(1 2 3 4)) ⇒ (((1 2) 3) 4)
(reduce #'list '(1 2 3 4) :from-end t) ⇒ (1 (2 (3 4)))
(reduce #'list '(1 2 3 4) :initial-value 'foo)
  ⇒ ((((foo 1) 2) 3) 4)
(reduce #'list '(1 2 3 4)
        :from-end t :initial-value 'foo)
  ⇒ (1 (2 (3 (4 foo))))
```

Если функция *function* имеет побочные эффекта, можно положится на порядок вызовов функции так, как было продемонстрировано выше.

Имя «reduce» было позаимствовано из APL.

X3J13 voted in March 1988 to extend the `reduce` function to take an additional keyword argument named `:key`. As usual, this argument defaults to the identity function. The value of this argument must be a function that accepts at least one argument. This function is applied once to each element of the sequence that is to participate in the reduction operation, in the order implied by the `:from-end` argument; the values returned by this function are combined by the reduction *function*. However, the `:key` function is *not* applied to the `:initial-value` argument (if any).

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

14.3 Модификация последовательностей

Каждая из этих функций или модифицирует последовательность, или возвращает модифицированную копию.

[Функция] **fill** *sequence item* **&key** *:start :end*

Функция модифицирует последовательность, заменяя каждый элемент подпоследовательности, обозначенной с помощью параметров `:start` и `:end`, объектом *item*. *item* может быть любым Lisp'овыми объектом, подходящим для типа элементов последовательности *sequence*. Объект *item* сохраняется в всех указанных компонентах последовательности *sequence*, начиная с позиции `:start` (по-умолчанию равна 0) и заканчивая невключительно позицией `:end` (по-умолчанию

равна длине последовательности). `fill` возвращает изменённую последовательность. Например:

(setq x (vector 'a 'b 'c 'd 'e)) ⇒ #(a b c d e)
(fill x 'z `:start` 1 `:end` 3) ⇒ #(a z z d e)
and now x ⇒ #(a z z d e)
(fill x 'p) ⇒ #(p p p p p)
and now x ⇒ #(p p p p p)

[Функция] **replace** *sequence1 sequence2* **&key** *:start1 :end1 :start2 :end2*

Функция модифицирует последовательность *sequence1* копируя в неё элементы из последовательности *sequence2*. Элементы *sequence2* для принадлежать типу элементов *sequence1*. Подпоследовательность *sequence2*, указанная с помощью параметров `:start2` и `:end2`, копируется в подпоследовательность *sequence2*, указанную с помощью параметров `:start1` и `:end1`. Аргументы `:start1` и `:start2` по-умолчанию равны нулю. Аргументы `:end1` и `:end2` по-умолчанию `nil`, что означает длины соответствующих последовательностей. Если указанные подпоследовательности имеют не равные длины, тогда для наиболее короткая из них определяет копируемые элементы. Лишние элементы последовательностей функцией не обрабатываются. Количество копируемых элементов можно вычислить так:

(min (- *end1 start1*) (- *end2 start2*))

Значением функции `replace` является модифицированная последовательность *sequence1*.

Если обе последовательности равны (`eq`), и интервал для модификации перекрывается с исходным интервалом, тогда поведение такое, как если бы все исходные данные сохранялись во временном месте, а затем сохранялись в итоговый интервал. Однако, если последовательности не равны, но итоговый интервал пересекается с исходным (возможно, при использовании *соединённых* массивов), тогда после выполнения `replace`, итоговое содержимое не определено.

[Функция] **remove** *item sequence* **&key** *:from-end :test :test-not :start :end :count :key*
[Функция] **remove-if** *predicate sequence* **&key** *:from-end :start :end :count :key*
[Функция] **remove-if-not** *predicate sequence* **&key** *:from-end :start :end :count :key*

Результатом является последовательность того же типа, что и последовательность *sequence*. Однако, итоговая последовательность не будет содержать элементы в интервале `:start-:end`, которые удовлетворяли условию. Результат является копией входящей последовательности *sequence* без исключённых элементов. Неудалённые элементы сохраняются в таком же порядке.

Если указан аргумент `:count`, то удаляться будет только это количество. Если количество удаляемых элементов превышает параметр, тогда будут удалены только самые левые в количестве равном `:count`.

X3J13 voted in January 1989 to clarify that the `:count` argument must be either `nil` or an integer, and that supplying a negative integer produces the same behavior as supplying zero.

Если при использовании `:count` для параметра `:from-end` указано не-`nil` значение, тогда удаление элементов будет происходить справа в количестве `:count`. Например:

(remove 4 '(1 2 4 1 3 4 5)) ⇒ (1 2 1 3 5)
(remove 4 '(1 2 4 1 3 4 5) `:count` 1) ⇒ (1 2 1 3 4 5)
(remove 4 '(1 2 4 1 3 4 5) `:count` 1 `:from-end` t)
⇒ (1 2 4 1 3 5)
(remove 3 '(1 2 4 1 3 4 5) `:test` #'>) ⇒ (4 3 4 5)
(remove-if #'oddp '(1 2 4 1 3 4 5)) ⇒ (2 4 4)
(remove-if #'evenp '(1 2 4 1 3 4 5) `:count` 1 `:from-end` t)
⇒ (1 2 4 1 3 5)

Результат `remove` может быть *соединён* с исходной последовательностью. Также результат может быть равен `eq` исходной последовательности, если ни одного элементы не было удалено.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

[Функция] **delete** *item sequence* **&key** *:from-end :test :test-not :start :end :count :key*
[Функция] **delete-if** *predicate sequence* **&key** *:from-end :start :end :count :key*
[Функция] **delete-if-not** *predicate sequence* **&key** *:from-end :start :end :count :key*

Данная функция в отличие от `remove` модифицирует исходную последовательность. Результатом является последовательность того же типа, что и последовательность *sequence*. Однако, итоговая последовательность не будет содержать элементы в интервале `:start-:end`, которые удовлетворяли условию. Результат является копией входящей последовательности *sequence* без исключённых элементов. Неудалённые элементы сохраняются в таком же порядке. Последовательность *sequence* может быть модифицирована, поэтому результат может быть равен `eq` или нет исходной последовательности.

Если указан аргумент `:count`, то удаляться будет только это количество. Если количество удаляемых элементов превышает параметр, тогда будут удалены только самые левые в количестве равном `:count`.

X3J13 voted in January 1989 to clarify that the `:count` argument must be either `nil` or an integer, and that supplying a negative integer produces the same behavior as supplying zero.

Если при использовании `:count` для параметра `:from-end` указано не-`nil` значение, тогда удаление элементов будет происходить справа в количестве `:count`. Например:

(delete 4 '(1 2 4 1 3 4 5)) ⇒ (1 2 1 3 5)
(delete 4 '(1 2 4 1 3 4 5) `:count` 1) ⇒ (1 2 1 3 4 5)
(delete 4 '(1 2 4 1 3 4 5) `:count` 1 `:from-end` t)
⇒ (1 2 4 1 3 5)
(delete 3 '(1 2 4 1 3 4 5) `:test` #'>) ⇒ (4 3 4 5)
(delete-if #'oddp '(1 2 4 1 3 4 5)) ⇒ (2 4 4)
(delete-if #'evenp '(1 2 4 1 3 4 5) `:count` 1 `:from-end` t)
⇒ (1 2 4 1 3 5)

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

X3J13 voted in March 1989 to clarify the permissible side effects of certain operations. When the *sequence* is a list, `delete` is permitted to perform a `setf` on any part, *car* or *cdr*, of the top-level list structure of that list. When the *sequence* is an array, `delete` is permitted to alter the dimensions of the given array and to slide some of its elements into new positions without permuting them in order to produce the resulting array.

Furthermore, `(delete-if` ***predicate sequence*** `...)` is required to behave exactly like

```
(delete nil sequence
      :test #'(lambda (unused item)
                (declare (ignore unused))
                (funcall predicate item))
      ...)
```

[Функция] **remove-duplicates** *sequence* **&key** *:from-end :test :test-not :start :end :key*
[Функция] **delete-duplicates** *sequence* **&key** *:from-end :test :test-not :start :end :key*

Функция попарно сравнивает элементы последовательности *sequence*, и если они равны, тогда первый из них удаляется (если параметр `:from-end` равен истине, то удаляется последний). Результат является последовательностью того же типа, что и исходная, с удалёнными повторяющимися элементами. Порядок следования элементов в итоге такой же как в исходной последовательности.

`remove-duplicates` является не модифицирующей версией этой операции. Результат `remove-duplicates` может быть *соединён* с исходной последовательностью. Также результат может быть равен `eq` исходной последовательности, если ни одного элементы не было удалено.

`delete-duplicates` может модифицировать аргумент *sequence*.

Например:

```
(remove-duplicates '(a b c b d d e)) ⇒ (a c b d e)
(remove-duplicates '(a b c b d d e) :from-end t) ⇒ (a b c d e)
(remove-duplicates '((foo #\a) (bar #\%) (baz #\A))
                   :test #'char-equal :key #'cadr)
  ⇒ ((bar #\%) (baz #\A))
```

```
(remove-duplicates '((foo #\a) (bar #\%) (baz #\A))
                   :test #'char-equal :key #'cadr :from-end t)
  ⇒ ((foo #\a) (bar #\%))
```

Эти функции полезны для преобразования последовательности в каноническую форму представления множества.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

X3J13 voted in March 1989 to clarify the permissible side effects of certain operations. When the *sequence* is a list, `delete-duplicates` is permitted to perform a `setf` on any part, *car* or *cdr*, of the top-level list structure of that list. When the *sequence* is an array, `delete-duplicates` is permitted to alter the dimensions of the given array and to slide some of its elements into new positions without permuting them in order to produce the resulting array.

[Функция] **substitute** *newitem olditem sequence* **&key** *:from-end :test :test-not :start :end :count :key*
[Функция] **substitute-if** *newitem test sequence* **&key** *:from-end :start :end :count :key*
[Функция] **substitute-if-not** *newitem test sequence* **&key** *:from-end :start :end :count :key*

Результатом является последовательность такого же типа что и исходная *sequence* за исключением того, что удовлетворяющие условию элементы в интервале `:start-:end` будут заменены на объект *newitem*. Эта операция создаёт копию исходной последовательности с некоторыми изменёнными элементами.

Если указан аргумент `:count`, то изменяться будет только это количество элементов. Если количество изменяемых элементов превышает параметр, тогда будут удалены только самые левые в количестве равном `:count`.

X3J13 voted in January 1989 to clarify that the `:count` argument must be either `nil` or an integer, and that supplying a negative integer produces the same behavior as supplying zero.

Если при использовании `:count` для параметра `:from-end` указано не-`nil` значение, тогда изменение элементов будет происходить справа в количестве `:count`. Например:

(substitute 9 4 '(1 2 4 1 3 4 5)) ⇒ (1 2 9 1 3 9 5)
(substitute 9 4 '(1 2 4 1 3 4 5) `:count` 1) ⇒ (1 2 9 1 3 4 5)
(substitute 9 4 '(1 2 4 1 3 4 5) `:count` 1 `:from-end` t)
⇒ (1 2 4 1 3 9 5)
(substitute 9 3 '(1 2 4 1 3 4 5) `:test` #'>) ⇒ (9 9 4 9 3 4 5)
(substitute-if 9 #'oddp '(1 2 4 1 3 4 5)) ⇒ (9 2 4 9 9 4 9)
(substitute-if 9 #'evenp '(1 2 4 1 3 4 5) `:count` 1 `:from-end` t)
⇒ (1 2 4 1 3 9 5)

Результат `substitute` может быть *соединён* с исходной последовательностью. Также результат может быть равен `eq` исходной последовательности, если ни одного элементы не было изменено.

Смотрите также `subst`, которая осуществляет замену в древовидной структуре.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

[Функция] **nsubstitute** *newitem olditem sequence* **&key** *:from-end :test :test-not :start :end :count :key*
[Функция] **nsubstitute-if** *newitem test sequence* **&key** *:from-end :start :end :count :key*
[Функция] **nsubstitute-if-not** *newitem test sequence* **&key** *:from-end :start :end :count :key*

Эта функция является деструктивным аналогом для `substitute`. Это значит, что они модифицирует свой аргумент. Результатом является последовательность такого же типа что и исходная *sequence* за исключением того, что удовлетворяющие условию элементы в интервале `:start-:end` будут заменены на объект *newitem*. Последовательность *sequence* может быть модифицирована, поэтому результат может быть равен `eq` или нет исходной последовательности.

Смотрите также `nsubst`, которая осуществляет деструктивную замену в древовидной структуре.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

X3J13 voted in March 1989 to clarify the permissible side effects of certain operations. When the *sequence* is a list, `nsubstitute` or `nsubstitute-if` is required to perform a `setf` on any *car* of the top-level list structure of that list whose old contents must be replaced with *newitem* but is forbidden

to perform a `setf` on any `cdr` of the list. When the *sequence* is an array, `nsubstitute` or `nsubstitute-if` is required to perform a `setf` on any element of the array whose old contents must be replaced with *newitem*. These functions, therefore, may successfully be used solely for effect, the caller discarding the returned value (though some programmers find this stylistically distasteful).

14.4 Поиск элементов последовательностей

Каждая из этих функций ищет в последовательности элемент, удовлетворяющий некоторому условию.

[Функция] **find** *item sequence* **&key** *:from-end :test :test-not :start :end :key*
[Функция] **find-if** *predicate sequence* **&key** *:from-end :start :end :key*
[Функция] **find-if-not** *predicate sequence* **&key** *:from-end :start :end :key*

Если последовательность *sequence* содержит элемент, удовлетворяющий условию, тогда возвращается первый найденный слева элемент, иначе возвращается `nil`.

Если заданы параметры `:start` и `:end`, тогда поиск осуществляется в этом интервале.

Если `:from-end` указан в не-`nil`, тогда результатом функции будет найденный элемент справа.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

[Функция] **position** *item sequence* **&key** *:from-end :test :test-not :start :end :key*
[Функция] **position-if** *predicate sequence* **&key** *:from-end :start :end :key*
[Функция] **position-if-not** *predicate sequence* **&key** *:from-end :start :end :key*

Если последовательность *sequence* содержит элемент, удовлетворяющий условию, тогда возвращается позиция найденного элемента слева, иначе возвращается `nil`.

Если заданы параметры `:start` и `:end`, тогда поиск осуществляется в этом интервале. Однако возвращаемый индекс будет относиться ко всей последовательности в целом.

Если `:from-end` указан в не-`nil`, тогда результатом функции будет индекс найденного элемент справа. (Однако, возвращаемый индекс, будет, как обычно, принадлежать нумерации слева направо.)

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

Ниже приведены несколько примеров использования нескольких функций, преимущественно `position-if` и `find-if`, для обработки строк. Также используется `loop`.

```
(defun debug-palindrome (s)
  (flet ((match (x) (char-equal (first x) (third x))))
    (let* ((pairs (loop for c across s
                        for j from 0
                        when (alpha-char-p c)
                          collect (list c j)))
           (quads (mapcar #'append pairs (reverse pairs)))
           (diffpos (position-if (complement #'match) quads)))
      (when diffpos
        (let* ((diff (elt quads diffpos))
               (same (find-if #'match quads
                              :start (+ diffpos 1))))
          (if same
              (format nil
                      "/~A/ (at ~D) is not the reverse of /~A/"
                      (subseq s (second diff) (second same))
                      (second diff)
                      (subseq s (+ (fourth same) 1)
                               (+ (fourth diff) 1)))
              "This palindrome is completely messed up!"))))))
```

Вот пример поведения этой функции

```
(setq panama     ;Предполагаемый палиндром?
      "A man, a plan, a canoe, pasta, heros, rajahs,
       a coloratura, maps, waste, percale, macaroni, a gag,
       a banana bag, a tan, a tag, a banana bag again
       (or a camel), a crepe, pins, Spam, a rut, a Rolo,
       cash, a jar, sore hats, a peon, a canal–Panama!")



(debug-palindrome panama)
 ⇒ "/wast/ (at 73) is not the reverse of /, pins/"

(replace panama "snipe" :start1 73)     ;Восстановить
 ⇒ "A man, a plan, a canoe, pasta, heros, rajahs,
      a coloratura, maps, snipe, percale, macaroni, a gag,
      a banana bag, a tan, a tag, a banana bag again
      (or a camel), a crepe, pins, Spam, a rut, a Rolo,
      cash, a jar, sore hats, a peon, a canal–Panama!"

(debug-palindrome panama) ⇒ nil     ;Первоклассный—истинный палиндром

(debug-palindrome "Rubber baby buggy bumpers")
 ⇒ "/Rubber / (at 0) is not the reverse of /umpers/"

(debug-palindrome "Common Lisp: The Language")
 ⇒ "/Commo/ (at 0) is not the reverse of /guage/"

(debug-palindrome "Complete mismatches are hard to find")
 ⇒
 "/Complete mism/ (at 0) is not the reverse of /re hard to find/"

(debug-palindrome "Waltz, nymph, for quick jigs vex Bud")
 ⇒ "This palindrome is completely messed up!"

(debug-palindrome "Doc, note: I dissent.  A fast never
                   prevents a fatness.  I diet on cod.")
 ⇒nil     ;Другой победитель
```

(debug-palindrome "Top step's pup's pet spot") ⇒ nil

[Функция] **count** *item sequence* **&key** *:from-end :test :test-not :start :end :key*
[Функция] **count-if** *predicate sequence* **&key** *:from-end :start :end :key*
[Функция] **count-if-not** *predicate sequence* **&key** *:from-end :start :end :key*

Результатом является неотрицательное целое число, указывающее на количество элементов из указанной подпоследовательности, удовлетворяющих условию.

`:from-end` не оказывает никакого воздействия, и оставлен только для совместимости с другими функциями.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

[Функция] **mismatch** *sequence1 sequence2* **&key** *:from-end :test :test-not :key :start1 :start2 :end1 :end2*

Функция поэлементно сравнивает указанные подпоследовательности. Если их длины равны и соответственно равны между собой каждые их элементы, то функция возвращает `nil`. Иначе, функция возвращает неотрицательное целое число. Это число указывает на первый элемент слева, который не совпал с элементом другой последовательности *sequence2*. Или же, в случае если одна подпоследовательность короче другой, но все элементы были одинаковыми, индекс указывает на последний проверенный элемент.

Если параметр `:from-end` равен не-`nil` значению, тогда возвращается увеличенная на 1 позиция первого не совпавшего элемента справа. Фактически, (под)последовательности выравниваются по своим правым концам, затем сравниваются последние элементы, затем предпоследние и так далее. Возвращаемый индекс принадлежит первой последовательности *sequence1*.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

[Функция] **search** *sequence1 sequence2* **&key** *:from-end :test :test-not :key :start1 :start2 :end1 :end2*

Функция осуществляет поэлементный поиск последовательности *sequence1* в последовательности *sequence1*. Если поиск не увенчался

успехом, результатом является значение `nil`. Иначе, результат является индексом первого совпавшего элемента слева во второй последовательности *sequence2*.

Если аргумент `:from-end` равен не-`nil`, поиск осуществляется справа, и результат является индексом элемента слева в первой совпавшей подпоследовательности справа.

Реализация может выбирать алгоритм поиска на своё усмотрение. Количество проводимых сравнений не может быть указано точно. Например, `search` при `:from-end` равном не-`nil` может фактически производить поиск слева направо, но результатом будет всегда индекс самой правой совпавшей последовательности. Поэтому пользователю желательно использовать предикаты без побочных эффектов.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

14.5 Сортировка и слияние

Эти функции могут модифицировать исходные данные: сортировать или соединять две уже отсортированные последовательности.

[Функция] **sort** *sequence predicate* **&key** *:key*
[Функция] **stable-sort** *sequence predicate* **&key** *:key*

Функция сортирует последовательность *sequence* в порядке, определяемом предикатом *predicate*. Результат помещается в исходную последовательность. Предикат *predicate* должен принимать два аргумента, возвращать не-`nil` тогда и только тогда, когда первый аргумент строго меньше второго (в подходящем для этого смысле). Если первый аргумент больше или равен второму (в подходящем для этого смысле), тогда предикат *predicate* должен вернуть `nil`.

Функция `sort` устанавливает отношение между двумя элементами с помощью предиката *predicate*, применённого к извлечённой из элемента ключевой части. Функция `:key`, применённая к элементу, должна возвращать его ключевую часть. Аргумент `:key` по-умолчанию равен функции эквивалентности, тем самым возвращая весь элемент.

Функция `:key` не должна иметь побочных эффектов. Полезным примером функции `:key` может быть функция-селектор для некоторой

структуры (созданной с помощью `defstruct`), используемая для сортировки последовательности структур.

(sort *a p* `:key` *s*) ≡ (sort *a* #'(lambda (x y) (*p* (*s* x) (*s* y))))

Тогда как оба выражения эквивалентны, для некоторых реализаций и определённых типов первое выражение может оказаться эффективнее. Например, реализация может выбрать сначала вычислить все ключевые части, поместить их в таблицу, а затем параллельно сортировать таблицы.

Если функции `:key` и *predicate* всегда возвращают управление, тогда операция сортировки будет всегда возвращать последовательность, содержащую такое же количество элементов как и в исходной (результат является просто последовательностью с переставленными элементами). Это поведение гарантируется, даже если предикат *predicate* в действительности не производит сравнение (в таком случае элементы будут расположены в неопределённом порядке, но ни один из них не будет потерян). Если функция `:key` последовательно возвращает необходимые ключевые части, и *predicate* отображает некоторый критерий для упорядочивания данных частей, тогда элементы итоговой последовательности будут корректно отсортированы в соответствие с этим критерием.

Операция сортировки, выполняемая с помощью `sort`, не гарантированно *постоянна*. Элементы, рассматриваемые предикатом *predicate* как равные, могут остаться или нет в оригинальном порядке. (Предполагается, что *predicate* рассматривает два элемента *x* и *y* равными, если `(funcall` ***predicate x y***`)` и `(funcall` ***predicate y x***`)` являются ложными.) Функция `stable-sort` гарантирует постоянность, но в некоторых ситуациях может быть медленнее чем `sort`.

Операция сортировки может быть деструктивное во всех случаях. В случае аргумента массива, она производится перестановкой элементов. В случае аргумента списка, список деструктивно переупорядочивается похожим образом как в `nreverse`. Таким образом, если аргумент не должен быть изменён, пользователь должен сортировать копию исходной последовательности.

Если применение функций `:key` или *predicate* вызывает ошибку, то состояние сортируемого массива или списка не определено. Однако,

если ошибка может быть исправлена, сортировка, конечно же, будет завершена корректно.

Следует отметить, что сортировка требует много сравнений и следовательно много вызовов предиката *predicate*, сортировка будет более быстрой, если *predicate* является компилируемой, а не интерпретируемой функцией.

Например:

```
(setq foovector (sort foovector #'string-lessp :key #'car))
```

Если `foovector` содержит эти данные перед сортировкой

```
("Tokens" "The Lion Sleeps Tonight")
("Carpenters" "Close to You")
("Rolling Stones" "Brown Sugar")
("Beach Boys" "I Get Around")
("Mozart" "Eine Kleine Nachtmusik" (K 525))
("Beatles" "I Want to Hold Your Hand")
```

тогда после неё, `foovector` будет содержать

```
("Beach Boys" "I Get Around")
("Beatles" "I Want to Hold Your Hand")
("Carpenters" "Close to You")
("Mozart" "Eine Kleine Nachtmusik" (K 525))
("Rolling Stones" "Brown Sugar")
("Tokens" "The Lion Sleeps Tonight")
```

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

[Функция] **merge** *result-type sequence1 sequence2 predicate* **&key** *:key*

Функция деструктивно соединяет две последовательности *sequence1* и *sequence2* в порядке определяемом предикатом *predicate*. Результат является последовательностью типа *result-type*, который должен быть подтипом `sequence`. Предикат *predicate* должен принимать два аргумента, возвращать не-`nil` тогда и только тогда, когда первый

аргумент строго меньше второго (в подходящем для этого смысле). Если первый аргумент больше или равен второму (в подходящем для этого смысле), тогда предикат *predicate* должен вернуть `nil`.

Функция `merge` устанавливает отношение между двумя элементами с помощью предиката *predicate*, применённого к извлечённой из элемента ключевой части. Функция `:key`, применённая к элементу, должна возвращать его ключевую часть. Аргумент `:key` по-умолчанию равен функции эквивалентности, тем самым возвращая весь элемент.

Функция `:key` не должна иметь побочных эффектов. Полезным примером функции `:key` может быть функция-селектор для некоторой структуры (созданной с помощью `defstruct`), используемая для сортировки последовательности структур.

Если функции `:key` и *predicate* всегда возвращают управление, тогда операция сортировки будет всегда завершаться. Результатом слияния двух последовательностей x и y является новая последовательность z, у которой длина равняется сумма длин x и y. z содержит все элементы x и y. Если *x1* и *x2* являются элементами x, и *x1* стоит прежде *x2*, тогда в z *x1* также будет стоять прежде чем *x2*. Такое же правило и для y. Если коротко, z является *слиянием* x и y.

Более того, если x и y были правильно отсортированы в соответствие с предикатом *predicate*, тогда z будет также правильно отсортирована. Например,

(merge 'list '(1 3 4 6 7) '(2 5 8) #'<) ⇒ (1 2 3 4 5 6 7 8)

Если x или y не были отсортированы, тогда z также не будет отсортирована. Однако слияние в любом случае произойдёт.

Операция слияния гарантированно *постоянна*. Если два или более элементов рассматриваются предикатом *predicate* как равные, тогда в результате элементы из *sequence1* будут предшествовать элементам из *sequence2*. (Предполагается, что *predicate* рассматривает два элемента x и y равными, если `(funcall` ***predicate x y***`)` и `(funcall` ***predicate y x***`)` являются ложными.) Например:

(merge 'string "BOY" "nosy" #'char-lessp) ⇒ "BnOosYy"

Результат не может быть `"BnoOsYy"`, `"BnOosyY"` или `"BnoOsyY"`. Функция `char-lessp` игнорирует регистр, таким образом, например, символы `Y` и `y` равны. Свойство постоянности гарантирует, что символ из первого аргумента (`Y`) должен предшествовать символу из второго аргумента (`y`).

X3J13 voted in June 1989 to specify that `merge` should signal an error if the sequence type specifies the number of elements and the sum of the lengths of the two sequence arguments is different.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

Глава 15

Списки

cons-ячейка, или пара с точкой, является составным объектом данных, содержащим два элемента, называемых *car* и *cdr*. Каждый компонент может является любым Lisp'овым объектом. *Список (list)* является цепочкой cons-ячеек, соединённых *cdr* элементами. Цепочка завершается некоторым не-cons объектом (atom object). Обычный список завершается объектом `nil`, или, как его ещё называют, пустым списком (). Список, цепочка которого завершается некоторым не-`nil` объектом, называется *списком с точкой*.

Для проверки конца списка служит предикат `endp`.

15.1 Cons-ячейки

Ниже представлены несколько основных операций над cons-ячейками. В данных случаях cons-ячейки рассматриваются как пары, а не компоненты списка.

[Функция] **car** *list*

Функция возвращает *car* элемент списка *list*, который должен быть или *cons* ячейкой или (). То есть аргумент должен удовлетворять предикату `listp`. По определению *car* элемент пустого списка является пустым списком. Если cons-ячейка является первым звеном списке, то можно сказать, что `car` возвращает первый элемент списка. Например:

(car '(a b c)) ⇒ a

Смотрите `first`.

car элемент cons-ячейки может быть изменён с помощью `rplaca` или `setf`.

[Функция] **cdr** *list*

Функция возвращает *cdr* элемент списка *list*, который должен быть или *cons* ячейкой или (). То есть аргумент должен удовлетворять предикату `listp`. По определению *cdr* элемент пустого списка является пустым списком. Если cons-ячейка является первым звеном списке, то можно сказать, что `cdr` возвращает остаток исходного списка без первого элемента. Например:

(cdr '(a b c)) ⇒ (b c)

Смотрите `rest`.

cdr элемент cons-ячейки может быть изменён с помощью `rplacd` или `setf`.

[Функция] **caar** *list*
[Функция] **cadr** *list*
[Функция] **cdar** *list*
[Функция] **cddr** *list*
[Функция] **caaar** *list*
[Функция] **caadr** *list*
[Функция] **cadar** *list*
[Функция] **caddr** *list*
[Функция] **cdaar** *list*
[Функция] **cdadr** *list*
[Функция] **cddar** *list*
[Функция] **cdddr** *list*
[Функция] **caaaar** *list*
[Функция] **caaadr** *list*
[Функция] **caadar** *list*
[Функция] **caaddr** *list*
[Функция] **cadaar** *list*
[Функция] **cadadr** *list*
[Функция] **caddar** *list*
[Функция] **cadddr** *list*
[Функция] **cdaaar** *list*
[Функция] **cdaadr** *list*
[Функция] **cdadar** *list*
[Функция] **cdaddr** *list*
[Функция] **cddaar** *list*
[Функция] **cddadr** *list*
[Функция] **cdddar** *list*
[Функция] **cddddr** *list*

Все композиции до четырёх `car` и `cdr` операций определены как самостоятельные функции. Их имена начинаются с `c` и заканчиваются `r`. Середина имени содержит последовательность букв `a` и `d` в соответствие с композиций выполняемых этими функциями. Например:

(cddadr x) is the same as (cdr (cdr (car (cdr x))))

Если в качестве аргумента указан список, тогда `cadr` вернёт второй элемент списка, `caddr` — третий, и `cadddr` — четвёртый. Если первый

элемент списка является списком, тогда `caar` вернёт первый элемент этого подсписка, `cdar` — остаток подсписка без первого элемента, `cadar` — второй элемент подсписка и так далее.

В целях стиля, предпочтительнее определить функцию или макрос для доступа к части сложной структуры данных, а не использовать длинные `car`/`cdr` строки. Например, можно определить макрос для получения списка параметров лямбда-выражения:

```
(defmacro lambda-vars (lambda-exp) `(cadr ,lambda-exp))
```

и затем использовать `lambda-vars` вместо `cadr`. Смотрите также `defstruct`, которая будет автоматически определять новые типы данных и функции доступа к частям экземпляров этих структур.

Любая из этих функций может использоваться в связке с `setf`.

[Функция] **cons** *x y*

`cons` является (примитивной) функцией для создания новой *cons*-ячейки, у которой *car* элемент будет *x*, а *cdr* элемент — *y*. Например:

```
(cons 'a 'b) ⇒ (a . b)
(cons 'a (cons 'b (cons 'c '()))) ⇒ (a b c)
(cons 'a '(b c d)) ⇒ (a b c d)
```

`cons` может рассматриваться как для создания *cons*-ячейки, так и для добавления нового элемента в начало списка.

[Функция] **tree-equal** *x y* **&key** *:test :test-not*

Данный предикат истинен, если *x* и *y* являются изоморфными деревьями с идентичными листьями, то есть, если *x* и *y* являются атомами, которые удовлетворяют проверке (по-умолчанию `eql`), или если они оба являются cons-ячейками и их *car* элементы удовлетворяют `tree-equal` и *cdr* элементы удовлетворяют `tree-equal`. Таким образом `tree-equal` рекурсивно сравнивает cons-ячейки (но не любой другой составной объект). Смотрите `equal`, который рекурсивно сравнивает другие составные объекты, как например строки.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

15.2 Списки

Следующие функции выполняет различные операции над списками.

Список является одним из первых Lisp'овых типов данных. Имя «Lisp» расшифровывается как «LISt Processing».

[Функция] **endp** *object*

Предикат `endp` используется для проверки конца списка. Возвращает ложь для cons-ячеек, истину для `nil`, и генерирует ошибку для всех остальных объектов других типов.

Заметка для реализации: Implementations are encouraged to signal an error, especially in the interpreter, for a non-list argument. The `endp` function is defined so as to allow compiled code to perform simply an atom check or a null check if speed is more important than safety.

[Функция] **list-length** *list*

`list-length` возвращает длину списка *list*. `list-length` отличается от `length` при использовании с циклическим списком. В таком случае `length` может не вернуть управление, тогда как `list-length` вернёт `nil`. Например:

```
(list-length '()) ⇒ 0
(list-length '(a b c d)) ⇒ 4
(list-length '(a (b c) d)) ⇒ 3
(let ((x (list 'a b c)))
  (rplacd (last x) x)
  (list-length x)) ⇒ nil
```

`list-length` может быть реализован так:

```
(defun list-length (x)
  (do ((n 0 (+ n 2))           ;Счётчик
       (fast x (cddr fast))    ;Быстрый указатель: на две позиции вперёд
       (slow x (cdr slow)))    ;Медленный указатель: на одну позицию
      (nil)
    ;; Если быстрый указатель дошёл до конца, вернуть длину.
    (when (endp fast) (return n))
```

```
(when (endp (cdr fast)) (return (+ n 1)))
;; If fast pointer eventually equals slow pointer,
;; then we must be stuck in a circular list.
;; (A deeper property is the converse: if we are
;; stuck in a circular list, then eventually the
;; fast pointer will equal the slow pointer.
;; That fact justifies this implementation.)
(when (and (eq fast slow) (> n 0)) (return nil))))
```

Смотрите `length`, которая возвращает длину любой последовательности.

[Функция] **nth** *n list*

`(nth n list)` возвращает n-нный элемент списка *list*. *car* элемент списка принимается за «нулевой» элемент. Аргумент n должен быть неотрицательным целым числом. Если длина списка не больше чем n, тогда результат (), или другими словами `nil`. (Это согласовывается с концепцией того, что *car* и *cdr* от () являются ().) Например:

```
(nth 0 '(foo bar gack)) ⇒ foo
(nth 1 '(foo bar gack)) ⇒ bar
(nth 3 '(foo bar gack)) ⇒ ()
```

`nth` может быть использован в связке с `setf` для изменения элемента списка. В этом случае, аргумент n должен быть меньше чем длина списка *list*.

Следует отметить, что порядок аргументов в `nth` обратный в отличие от большинства других функций селекторов для последовательностей, таких ка `elt`.

[Функция] **first** *list*
[Функция] **second** *list*
[Функция] **third** *list*
[Функция] **fourth** *list*
[Функция] **fifth** *list*
[Функция] **sixth** *list*
[Функция] **seventh** *list*
[Функция] **eighth** *list*
[Функция] **ninth** *list*
[Функция] **tenth** *list*

Иногда эти функции удобно использовать для доступа к определёнными элементам списка. `first` то же, что и `car`, `second` то же, что и `cadr`, `third` то же, что и `caddr`, и так далее. Следует отметить, что нумерация начинается с единицы (first) в отличие от нумерации, которая начинается с нуля и используется в `nth`.

(fifth x) ≡ (nth 4 x)

Каждая из этих функций может быть использована в связке `setf` для изменения элемента массива.

[Функция] **rest** *list*

`rest` означает то же, что и `cdr`, но мнемонически согласуется с `first`. `rest` может использоваться в связке с `setf` для изменения элементов массива.

[Функция] **nthcdr** *n list*

`(nthcdr` ***n list***`)` выполняет для списка *lisp* операцию `cdr` *n* раз, и возвращает результат. Например:

(nthcdr 0 '(a b c)) ⇒ (a b c)
(nthcdr 2 '(a b c)) ⇒ (c)
(nthcdr 4 '(a b c)) ⇒ ()

Другими словами, она возвращает n-нную *cdr* часть списка.

(car (nthcdr n x)) ≡ (nth n x)

Аргумент n должен быть неотрицательным целым числом.

[Функция] **last** *list* **&optional** (n 1)

`last` возвращает последние n cons-ячеек списка *lisp*. Список *list* может быть списком с точкой. Передача зацикленного списка является ошибкой.

Аргумент n должен быть неотрицательным целым числом. Если n равен нулю, тогда возвращается последний атом списка *list*. Если n не меньше чем количество cons-ячеек, то возвращается весь список.

Например:

(setq x '(a b c d))
(last x) ⇒ (d)
(rplacd (last x) '(e f))
x ⇒ '(a b c d e f)
(last x 3) ⇒ (d e f)
(last '()) ⇒ ()
(last '(a b c . d)) ⇒ (c . d)
(last '(a b c . d) 0) ⇒ d
(last '(a b c . d) 2) ⇒ (b c . d)
(last '(a b c . d) 1729) ⇒ (a b c . d)

[Функция] **list** **&rest** *args*

`list` создаёт и возвращает список, составленный из аргументов. Например:

(list 3 4 'a (car '(b . c)) (+ 6 -2)) ⇒ (3 4 a b 4)

(list) ⇒ ()
(list (list 'a 'b) (list 'c 'd 'e)) ⇒ ((a b) (c d e))

[Функция] **list*** *arg* **&rest** *others*

`list*` похожа на `list` за исключением того, что последняя *cons*-ячейка создаваемого списка будет «с точкой». Последний аргумент используется как последний элемент списка, а именно в последней cons-ячейки в *cdr* элементе. Данный аргумент необязательно должен быть атомом, и если он не атом, то в результате список будет иметь большую длину чем количество аргументов. Например:

(list* 'a 'b 'c 'd) ⇒ (a b c . d)

Это то же, что и

(cons 'a (cons 'b (cons 'c 'd)))

А также:

(list* 'a 'b 'c '(d e f)) ⇒ (a b c d e f)
(list* x) ≡ x

[Функция] **make-list** *size* **&key** *:initial-element*

Функция создаёт и возвращает список содержащий количество *size* элементов, каждый из которых будет инициализирован значением аргумента `:initial-element` (который по-умолчанию `nil`). *size* должен быть неотрицательным целым числом. Например:

(make-list 5) ⇒ (`nil nil nil nil nil`)
(make-list 3 `:initial-element` 'rah) ⇒ (rah rah rah)

[Функция] **append** **&rest** *lists*

Функция возвращает список содержащий все элементы указанных в аргументах списков. Аргументы не разрушаются. Например:

(append '(a b c) '(d e f) '() '(g)) ⇒ (a b c d e f g)

Следует отметить, что `append` копирует верхний уровень всех переданных списков *за исключением* последнего. Функция `concatenate` выполняет похожую операцию, но всегда копирует все аргументы. Смотрите также `nconc`, которая похожа на `append`, но разрушает все аргументы кроме последнего.

Последний аргумент может быть любым Lisp объектом, и в этом случае этот объект становится последним элементов итогового списка. Например, `(append '(a b c) 'd)` ⇒ `(a b c . d)`.

`(append` *x* `'())` может быть использовано для копирования списка *x*, однако для этого больше подходит функция `copy-list`.

[Функция] **copy-list** *list*

Функция возвращает список, который равен `equal` и в то же время не равен `eq` списку *list*, Копируется только верхний уровень списка, то есть `copy-list` копирует только в направлении *cdr* элементов, но не в направлении *car* элементов. Если список «с точкой», то есть `(cdr (last` *list*`))` является не-`nil` атомом, тогда итоговый список также будет «с точкой». Смотрите также `copy-seq` и `copy-tree`.

[Функция] **copy-alist** *list*

`copy-alist` копирует ассоциативные списки. При этом, также как и в `copy-list`, копируется только верхний уровень списка *lisp*. Кроме того, каждый элемент списка *list*, являющийся в свою очередь cons-ячейкой, заменяется новой cons-ячейкой с теми же *car* и *cons* элементами.

[Функция] **copy-tree** *object*

`copy-tree` копирует древовидно организованные cons-ячейки. Аргумент *object* может быть любым Lisp'овым объектом. Если он не является cons-ячейкой, то ничего не произойдёт и данный объект будет возвращён в качестве результата. В противном случае будет возвращена новая cons-ячейка, в которой *car* и *cons* элементы будут результатами рекурсивных вызовов `copy-tree`. Другими словами, все cons-ячейки будут рекурсивно скопированы, и рекурсия будет останавливаться только на атомах.

[Функция] **revappend** *x y*

`(revappend x y)` похожа на `(append (reverse x) y)` за исключением того, что она потенциально более производительна. Аргументы *x* и *y* должны быть списками. Аргумент *x* копируется (не разрушается) в отличие от `nreconc`, которая разрушает первый аргумент.

[Функция] **nconc** **&rest** *lists*

В качестве аргументов `nconc` принимает списки. Функция соединяет списки и возвращает результат. При этом аргументы изменяются, а не копируются. (В сравнении с `append`, которая копирует аргументы, а не разрушает их.) Например:

(setq x '(a b c))
(setq y '(d e f))
(nconc x y) ⇒ (a b c d e f)
x ⇒ (a b c d e f)

Следует отметить, что в примере, значение `x` отличается от первоначального, так как последняя cons-ячейка была изменена с помощью `rplacd` значением `y`. Если сейчас выполнить `(nconc x y)` ещё раз, тогда часть списка зациклится: `(a b c d e f d e f d e f ...)`, и так до бесконечности. Если `*print-circle*` не равен `nil`, тогда вывод списка будет таким: `(a b c . #1=(d e f . #1#))`.

Вызов `nconc`, совпадающий с наиболее близким шаблоном выражения в левой части приводит к эквивалентным побочным действиям, как в правой части таблицы.

```
(nconc)                 nil     ;Нет побочных эффектов
(nconc nil . r)         (nconc . r)
(nconc x)               x
(nconc x y)             (let ((p x) (q y))
                           (rplacd (last p) q)
                           p)
(nconc x y . r)         (nconc (nconc x y) . r)
```

[Функция] **nreconc** *x y*

`(nreconc x y)` похожа на `(nconc (nreverse x) y)` за исключением того, что она потенциально эффективнее. Оба аргумента должны быть списками. Аргумент *x* разрушается. Сравните с `revappend`.

```
(setq planets '(jupiter mars earth venus mercury))
(setq more-planets '(saturn uranus pluto neptune))
(nreconc more-planets planets)
        ⇒ (neptune pluto uranus saturn jupiter mars earth venus mercury)
 теперь значение more-planets точно не определено
```

Поведение `(nreconc x y)` совпадает с поведением `(nconc (nreverse x) y)` в части побочных эффектов.

[Макрос] **push** item place

Форма *place* должна быть именем обобщённое переменной, содержащей список. *item* может указывать на любой Lisp'овый объект. *item* вставляется в начало списка и данный список возвращается в качестве результата. Форма *place* может быть любой формой, которая подходит для `setf`. Если рассматривать список как стек, тогда `push` добавляет элемент на вершину стека. Например:

```
(setq x '(a (b c) d))
(push 5 (cadr x)) ⇒ (5 b c) и теперь x ⇒ (a (5 b c) d)
```

Действие от

(push *item place*)

эквивалентно действию

(setf *place* (cons *item place*))

за исключением того, что `push` выполняет форму *place* только один раз, а не три. Более того, в для некоторых форм *place* `push` может быть эффективнее чем версия с `setf`.

Следует отметить, что *item* вычисляется прежде чем вычисляется *place*.

[Макрос] **pushnew** item place &key :test :test-not :key

Форма *place* должна быть именем обобщённое переменной, содержащей список. *item* может указывать на любой Lisp'овый объект. Если *item* не содержится в списке (этот факт устанавливается с помощью предиката переданного в `:test`, который по-умолчанию `eql`), тогда *item* вставляется в начало списка и данный список возвращается в качестве результата. В противном случае возвращается исходный список. Форма *place* может быть любой формой, которая подходит для `setf`. Если рассматривать список как множество, тогда `pushnew` добавляет элемент в множество. Смотрите `adjoin`.

Именованные параметры `pushnew` имеют тот же смысл, что и в функциях для последовательностей. Смотрите главу 14. По сути, данные аргументы идентичны аргументам `adjoin`.

`pushnew` возвращает модифицированное содержимое переменной *place*. Например:

(setq x '(a (b c) d))
(pushnew 5 (cadr x)) ⇒ (5 b c) и теперь x ⇒ (a (5 b c) d)
(pushnew 'b (cadr x)) ⇒ (5 b c) и `x` не меняется

Действие от

(pushnew *item place* `:test` *p*)

эквивалентно действию

(setf *place* (adjoin *item place* `:test` *p*))

за исключением того, что `pushnew` выполняет форму *place* только один раз, а не три. Более того, в для некоторых форм *place* `pushnew` может быть эффективнее чем версия с `setf`.

Следует отметить, что *item* вычисляется прежде чем вычисляется *place*.

[Макрос] **pop** place

Форма *place* должна быть именем обобщённое переменной, содержащей список. Результатом `pop` является результат `car` функции для переданного списка, и побочным эффектом является то, что в обобщённую переменную сохраняется результат `cdr` для списка. Форма *place* может быть любой формой, которая подходит для `setf`. Если рассматривать исходный список как стек, то `pop` достаёт элемент из вершины стека и возвращает его. Например:

(setq stack '(a b c))
(pop stack) ⇒ a and now stack ⇒ (b c)

Действия от `(pop` *`place`*`)` эквивалентно

(prog1 (car *place*) (setf *place* (cdr *place*)))

за исключением того, что `pop` выполняет форму *place* только один раз, а не три. Более того, в для некоторых форм *place* `pop` может быть эффективнее чем версия с `setf`.

[Функция] **butlast** *list* **&optional** *n*

Функция создаёт и возвращает список с такими же элементами кроме *n* последних, что и в списке *list*. *n* по-умолчанию равно 1. Аргумент не разрушается. Если длина списка *list* меньше чем *n*, тогда возвращается (). Например:

(butlast '(a b c d)) ⇒ (a b c)
(butlast '((a b) (c d))) ⇒ ((a b))
(butlast '(a)) ⇒ ()
(butlast nil) ⇒ ()

Имя функции образовано от фразы «all elements but the last» («все элементы кроме последних»).

[Функция] **nbutlast** *list* **&optional** *n*

Это деструктивная версия `butlast`. Данная функция изменяет *cdr* элемент cons-ячейки на `nil`. Искомая cons-ячейка находится на позиции n+1 с конца списка. Если длина списка *list* меньше чем n, тогда возвращается (), и аргумент не модифицируется. (Таким образом можно написать `(setq a (nbutlast a))`, а не `(nbutlast a)`.) Например:

(setq foo '(a b c d))
(nbutlast foo) ⇒ (a b c)
foo ⇒ (a b c)
(nbutlast '(a)) ⇒ ()
(nbutlast '`nil`) ⇒ ()

[Функция] **ldiff** *list sublist*

Аргумент *list* должен быть списком, и *sublist* должен быть подсписком *list*. `ldiff` (означает «list difference») возвращает новый список, элементы которого содержат все элементы списка *list* до подсписка *sublist*. Если *sublist* не является частью *list* (или в частности равен `nil`), тогда возвращается копия всего списка *list*. Аргумент *list* не разрушается. Например:

(setq x '(a b c d e))
(setq y (cdddr x)) ⇒ (d e)
(ldiff x y) ⇒ (a b c)
но (ldiff '(a b c d) '(c d)) ⇒ (a b c d)

так как подсписок не равен `eq` ни одной части списка.

15.3 Изменение структуры списка

Для изменения структуры уже имеющегося списка могут использоваться функции `rplaca` и `rplacd`. Данные функции изменяют *car* и *cdr* элементы cons-ячеек соответственно. Можно также использовать `setf` в связке с `car` и `cdr`.

Структура списка деструктивно изменяется, а не копируется. Такое поведение может оказаться неожиданным, особенно при использовании частей списком, на которые указывают более одной переменной. Описанные ранее функции `nconc`, `nreverse`, `nreconc`, и `nbutlast` также деструктивно изменяют список. Данные функции имеют «копирующие, а не разрушающие» аналоги.

[Функция] **rplaca** *x y*

`(rplaca x y)` изменяет *car* элемент в cons-ячейке *x* на *y* и возвращает (модифицированную) *x*. *x* должен быть cons-ячейкой, но *y* может быть любым Lisp'овым объектом. Например:

(setq g '(a b c))
(rplaca (cdr g) 'd) ⇒ (d c)
Теперь g ⇒ (a d c)

[Функция] **rplacd** *x y*

`(rplacd x y)` изменяет *cdr* элемент в cons-ячейке *x* на *y* и возвращает (модифицированную) *x*. *x* должен быть cons-ячейкой, но *y* может быть любым Lisp'овым объектом. Например:

(setq x '(a b c))
(rplacd x 'd) ⇒ (a . d)
Теперь x ⇒ (a . d)

Функции `rplaca` и `rplacd` пришли из самых ранних версий Lisp'а, как `car`, `cdr` и `cons`. Однако, в настоящее время, они, похоже, отходят на второй план. Все больше и больше Common Lisp программистов используют `setf` почти для всех изменений структур: `(rplaca x y)` становится `(setf (car x) y)` или, возможно, `(setf (first x) y)`. Ещё

более вероятно, что структура `defstruct` или CLOS класс используют вместо списком, если структура данных слишком сложная. В таком случае `setf` используется в связке с функцией доступа к слоту.

15.4 Замещение выражений

Для выполнения операции замещения в древовидной структуре cons-ячеек предоставлен ряд функций. Все эти функции принимают дерево и описание того, что на что необходимо заменить. Функции имеют копирующие и деструктивные версии, а также версии в которых замещение описывается либо двумя аргументами, либо ассоциативным списком.

Правила именования для этих функций и для их именованных параметров совпадают с правилами функций для последовательностей. Смотрите раздел 14.

[Функция] **subst** *new old tree* **&key** *:test :test-not :key*
[Функция] **subst-if** *new test tree* **&key** *:key*
[Функция] **subst-if-not** *new test tree* **&key** *:key*

`(subst` *`new old tree`*`)` создаёт копию дерева *tree*, замещая элемент *old* элементом *new*. Замещение происходит в любом месте дерева. Функция возвращает модифицированную копию дерева *tree*. Исходный объект *tree* не изменяется, но итоговое дерево может иметь общие с исходным части.

Например:

```
(subst 'tempest 'hurricane
       '(shakespeare wrote (the hurricane)))
  ⇒ (shakespeare wrote (the tempest))

(subst 'foo 'nil '(shakespeare wrote (twelfth night)))
  ⇒ (shakespeare wrote (twelfth night . foo) . foo)

(subst '(a . cons) '(old . pair)
       '((old . spice) ((old . shoes) old . pair) (old . pair))
       :test #'equal)
```

⇒ ((old . spice) ((old . shoes) a . cons) (a . cons))

Эта функция не является деструктивной. Она не изменяет `car` и `cdr` элементы уже существующего дерева. Можно определить `subst` так:

```
(defun subst (old new tree &rest x &key test test-not key)
  (cond ((satisfies-the-test old tree :test test
                             :test-not test-not :key key)
         new)
        ((atom tree) tree)
        (t (let ((a (apply #'subst old new (car tree) x))
                 (d (apply #'subst old new (cdr tree) x)))
             (if (and (eql a (car tree))
                      (eql d (cdr tree)))
                 tree
                 (cons a d))))))
```

Смотрите также `substitute`, которая проводит замещение только для верхнего уровня списка.

[Функция] **nsubst** *new old tree* **&key** *:test :test-not :key*
[Функция] **nsubst-if** *new test tree* **&key** *:key*
[Функция] **nsubst-if-not** *new test tree* **&key** *:key*

`nsubst` является деструктивным аналогом `subst`. В дереве *tree* любой элемент *old* заменяется на *new*.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

[Функция] **sublis** *alist tree* **&key** *:test :test-not :key*

`sublis` выполняет замещение объектов в дереве (древовидной структуре из cons-ячеек). Первый аргумент `sublis` является ассоциативным списком. Второй аргумент — дерево, в котором выполняется замещение. `sublis` проходит по всему дереву включая листья, и если элемент встречается в качестве ключа в ассоциативном списке, то данный элемент заменяет на значение ключа. Данная операция не разрушает дерево. `sublis` может выполнять несколько `subst` операций за один раз. Например:

```
(sublis '((x . 100) (z . zprime))
        '(plus x (minus g z x p) 4 . x))
  ⇒ (plus 100 (minus g zprime 100 p) 4 . 100)

(sublis '(((+ x y) . (- x y)) ((- x y) . (+ x y)))
        '(* (/ (+ x y) (+ x p)) (- x y))
        :test #'equal)
  ⇒ (* (/ (- x y) (+ x p)) (+ x y))
```

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

[Функция] **nsublis** *alist tree* **&key** *:test :test-not :key*

`nsublis` похожа на `sublis` но деструктивно модифицирует необходимые элементы дерева *tree*.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

15.5 Использование списков как множеств

Common Lisp содержит функции, которые позволяют обрабатывать списки элементов как *множества*. Сюда входят функции добавления, удаления и поиска элементов в списке, основанного на различных критериях. Кроме того, включены функции объединения, пересечения и разности.

Правила наименования данных функций и их именованных параметров в основном следуют правилам именования функций для последовательностей. Смотрите главу 14.

[Функция] **member** *item list* **&key** *:test :test-not :key*
[Функция] **member-if** *predicate list* **&key** *:key*
[Функция] **member-if-not** *predicate list* **&key** *:key*

Функция осуществляет поиск элемента, удовлетворяющего условию, в списке *list*. Если элемент не найдёт, возвращается `nil`. Иначе возвращается часть списка, начинающаяся с искомого элемента. Поиск

осуществляется только в верхнем уровне списка. Эти функции могут использоваться в качестве предикатов.

Например:

(member 'snerd '(a b c d)) ⇒ `nil`
(member-if #'numberp '(a #\Space 5/3 foo)) ⇒ (5/3 foo)
(member 'a '(g (a y) c a d e a f)) ⇒ (a d e a f)

Следует отметить, что в последнем примере значение, возвращённое `member`, равно `eq` части списка, которая начинается на `a`. Если `member` вернула не `nil` значение, то для изменения полученного элемента списка можно использовать `rplaca`.

Смотрите также `find` и `position`.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

[Функция] **tailp** *sublist list*

`tailp` истинен тогда и только тогда, когда существует такое целое число *n*, что выполняется

(eql *sublist* (nthcdr *n list*))

list может быть списком с точкой (подразумевается, что реализации могут использовать `atom` и не могут `endp` для проверки конца списка *list*). FIXME

[Функция] **adjoin** *item list* **&key** *:test :test-not :key*

`adjoin` используется для добавления элементов во множество, если этого элемента во множестве ещё не было. Условие равенства по-умолчанию `eql`.

(adjoin *item list*) ≡ (if (member *item list*) *list* (cons *item list*))

Условие равенства может быть любым предикатом. *item* добавляется в список тогда и только тогда, когда в списке не было ни одного элемента, «удовлетворяющего условию».

`adjoin` отклоняется от обычных правил, описанных в главе 14 в части обработки параметров *item* и `:key`. Если указана `:key` функция, то она применяется к параметру *item*, также как и к каждому элементу списка. Обоснование в том, что если *item* ещё не был в списке и если он там появится, то применение функции `:key` к нему как элементу списка не будет корректным, если этого не было при его добавлении.

(adjoin *item list* :key *fn*)
≡ (if (member (funcall *fn item*) *list* :key *fn*) *list* (cons *item list*))

Смотрите также `pushnew`.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

[Функция] **union** *list1 list2* **&key** *:test :test-not :key*
[Функция] **nunion** *list1 list2* **&key** *:test :test-not :key*

`union` принимает два списка и возвращает новый список, содержащий всё, что является элементами списков *list1* и *list2*. Если в списках есть дубликаты, то в итоговом будет только один экземпляр. Например:

(union '(a b c) '(f a d))
⇒ (a b c f d) или (b c f a d) или (d f a b c) или ...

(union '((x 5) (y 6)) '((z 2) (x 4)) :key #'car)
⇒ ((x 5) (y 6) (z 2)) или ((x 4) (y 6) (z 2)) или ...

Порядок элементов в итоговом списке не обязательно совпадает с порядком соответствующих элементов в списках. Итоговый список может иметь общие ячейки с или быть равным `eq` переданным аргументам.

Функция `:test` может быть любым предикатом, и операция объединения может быть описана следующим образом. Для всех возможных упорядоченных пар, состоящих из одного элемента из списка *list1* и одного элемента из списка *list2*, предикат устанавливает «равны»

ли они. Для каждой пары равных элементов, как минимум один из двух элементов будет помещён в результат. Кроме того, любой элемент, которые не был равен ни одному другому элементу, также будет помещён в результат. Это описание может быть полезным при использовании хитрых функций проверки равенства.

Аргумент `:test-not` может быть полезен, когда функция проверки равенства является логическим отрицанием проверки равенства. Хороший пример такой функции это `mismatch`, которая логически инвертирована так, что если аргументы не равны, то может быть получена возможная полезная информация. Эта дополнительная «полезная информация» отбрасывается в следующем примере. `mismatch` используется только как предикат.

```
(union '(#(a b) #(5 0 6) #(f 3))
       '(#(5 0 6) (a b) #(g h))
       :test-not
       #'mismatch)
  ⇒ (#(a b) #(5 0 6) #(f 3) #(g h))     ;Возможный результат
  ⇒ ((a b) #(f 3) #(5 0 6) #(g h))      ;Другой возможный результат
```

Использование `:test-not #'mismatch` отличается от использования `:test #'equalp`, например, потому что `mismatch` определяет что `#(a b)` и `(a b)` одинаковы, тогда как `equalp` определяет эти выражения разными.

`nunion` является деструктивной версией `union`. Она выполняет ту же операцию, но может разрушить аргументы, возможно при использовании их ячеек для построения результата.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

X3J13 voted in March 1989 to clarify the permissible side effects of certain operations; `nunion` is permitted to perform a `setf` on any part, *car* or *cdr*, of the top-level list structure of any of the argument lists.

[Функция] **intersection** *list1 list2* **&key** *:test :test-not :key*
[Функция] **nintersection** *list1 list2* **&key** *:test :test-not :key*

`intersection` принимает два списка и возвращает новый список содержащий все элементы, которые есть и в первом и во втором списках одновременно. Например:

(intersection '(a b c) '(f a d)) ⇒ (a)

Порядок элементов в итоговом списке не обязательно совпадает с порядком соответствующих элементов в списках. Итоговый список может иметь общие ячейки с или быть равным `eq` переданным аргументам.

Функция `:test` может быть любым предикатом, и операция пересечения может быть описана следующим образом. Для всех возможных упорядоченных пар, состоящих из одного элемента из списка *list1* и одного элемента из списка *list2*, предикат устанавливает «равны» ли они. Для каждой пары равных элементов, только один из двух элементов будет помещён в результат. Больше никаких элементов в итоговом списке не будет. Это описание может быть полезным при использовании хитрых функций проверки равенства.

`nintersection` является является деструктивной версией `intersection`. Она выполняет ту же операцию, но может разрушить аргумент *list1*, возможно при использовании их ячеек для построения результата. (Аргумент *list2* не разрушается.)

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

X3J13 voted in March 1989 to clarify the permissible side effects of certain operations; `nintersection` is permitted to perform a `setf` on any part, *car* or *cdr*, of the top-level list structure of any of the argument lists.

[Функция] **set-difference** *list1 list2* **&key** *:test :test-not :key*
[Функция] **nset-difference** *list1 list2* **&key** *:test :test-not :key*

`set-difference` возвращает список элементов списка *list1*, которые не встречаются в списке *list2*. Данная операция не разрушает аргументы.

Порядок элементов в итоговом списке не обязательно совпадает с порядком соответствующих элементов в списке *list1*. Итоговый список может иметь общие ячейки, или быть равным `eq` аргументу *list1*.

:test может быть любым предикатом, и операция разности множеств может быть описана следующим образом. Для всех возможных упорядоченных пар, состоящих из элементов первого и второго списков, используется предикат для установки их «равенства». Элемент из списка *list1* помещается в результат, тогда и только тогда, когда он не равен ни одному элементу списка. Это позволяет делать очень интересные

приложения. Например, можно удалить из списка строк все строки, содержащие некоторый список символов: *list2*.

```
;; Удалить все имена специй содержащие буквы "c" или "w".
(set-difference '("strawberry" "chocolate" "banana"
                  "lemon" "pistachio" "rhubarb")
                '(#\c #\w)
                :test
                #'(lambda (s c) (find c s)))
  ⇒ ("banana" "rhubarb" "lemon")     ;Возможен другой порядок элементов
```

`nset-difference` является деструктивной версией `set-difference`. Данная операция может разрушить *list1*.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

[Функция] **set-exclusive-or** *list1 list2* **&key** *:test :test-not :key*
[Функция] **nset-exclusive-or** *list1 list2* **&key** *:test :test-not :key*

`set-exclusive-or` возвращает список элементов, которые встречаются только в списке *list1* и только в списке *list2*. Данная операция не разрушает аргументы.

Порядок элементов в итоговом списке не обязательно совпадает с порядком соответствующих элементов в списке *list1*. Итоговый список может иметь общие ячейки, или быть равным `eq` аргументу *list1*.

Функция проверки равенства элементов может быть любым предикатом, и операцию `set-exclusive-or` можно описать следующим образом. Для всех возможных упорядоченных пар, содержащих один элемент из списка *list1* и один элемент из списка *list2*, функция используется для проверки «равенства». Результат содержит точно те элементы списков *list1* и *list2*, которые были только в различающихся парах.

`nset-exclusive-or` является деструктивной версией `set-exclusive-or`. Данная операция может разрушить аргументы.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

X3J13 voted in March 1989 to clarify the permissible side effects of certain operations; `nset-exclusive-or` is permitted to perform a `setf` on any part, *car* or *cdr*, of the top-level list structure of any of the argument lists.

[Функция] **subsetp** *list1 list2* **&key** *:test :test-not :key*

`subsetp` является предикатом, который истинен, если каждый элемент списка *list1* встречается в («равен» некоторому элементу в) списке *list2*, иначе ложен.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

15.6 Ассоциативные списки

Ассоциативный список, или *a-list*, является структурой данных, часто используемой в Lisp'е. a-list является списком пар (cons-ячеек). Каждая пара является ассоциацией. *car* элемент пары называется *key*, и *cdr* элемент *datum*.

Преимущество a-list представления данных в том, что a-list может быть постепенно увеличен путём простого добавления в начало новых записей. Кроме того, поскольку функция поиска `assoc` по порядку проходит элементы a-list, то новые записи могут «скрыть» старые. Если a-list рассматривать как отображение ключей в значения, то отображение может быть не только увеличено, но также изменено с помощью добавления новых записей в начало a-list.

Иногда a-list представляет биективное (bijective) отображение, и бывает нужно получить ключ для некоторого значения. Для этих целей используется функция «реверсивного» поиска `rassoc`. Другие варианты поиска в a-list могут быть созданы с помощью функции `find` или `member`.

Допустимо, чтобы `nil` был элемент a-list вместо пары ключ-значение. Такой элемент не считается парой и просто пропускается при использовании функции `assoc`.

[Функция] **acons** *key datum a-list*

`acons` создаёт новый ассоциативный список, с помощью добавления пары `(`*`key`*` . `*`datum`*`)` к старому *a-list*.

(acons x y a) ≡ (cons (cons x y) a)

[Функция] **pairlis** *keys data* **&optional** *a-list*

`pairlis` принимает два списка и создаёт ассоциативный список, который связывает элементы первого списка с соответствующими элементами второго. Если списки не одинаковой длины, это является ошибкой. Если указан необязательный аргумент *a-list*, тогда новые пары добавляются к нему в начало.

Новые пары могут быть расположены в итоговом списке a-list в любом порядке. Например:

```
(pairlis '(one two) '(1 2) '((three . 3) (four . 19)))
```

может быть

```
((one . 1) (two . 2) (three . 3) (four . 19))
```

или может быть так

```
((two . 2) (one . 1) (three . 3) (four . 19))
```

[Функция] **assoc** *item a-list* **&key** *:test :test-not :key*
[Функция] **assoc-if** *predicate a-list* **&key** *:key*
[Функция] **assoc-if-not** *predicate a-list* **&key** *:key*

Каждая из этих функций осуществляет поиск в ассоциативном списке *a-list*. Функция возвращает первую пару, удовлетворяющую условию, или `nil`, если такой пары не было найдено. Например:

```
(assoc 'r '((a . b) (c . d) (r . x) (s . y) (r . z)))
      ⇒ (r . x)
(assoc 'goo '((foo . bar) (zoo . goo))) ⇒ nil
(assoc '2 '((1 a b c) (2 b c d) (-7 x y z))) ⇒ (2 b c d)
```

Если функция вернула пару, можно изменить её значение с помощью `rplacd`. (Однако лучше будет добавить новую «затеняющую» пару в начало, чем модифицировать старую.) Например:

(setq values '((x . 100) (y . 200) (z . 50)))
(assoc 'y values) ⇒ (y . 200)
(rplacd (assoc 'y values) 201)
(assoc 'y values) ⇒ (y . 201) теперь

Типичный приём использования `(cdr (assoc x y))`. Так как *cdr* от `nil` гарантировано вычисляется в `nil`, то в случае отсутствия нужной пары, будет возвращено значение `nil`. `nil` также будет возвращено, если значение для ключа равно `nil`. Такое поведение удобно, если `nil` несёт смысл «значения по-умолчанию».

Два выражения

(assoc *item list* :test *fn*)

и

(find *item list* :test *fn* :key #'car)

эквивалентны за исключением, того что `assoc` игнорирует значения `nil` на месте пар.

Смотрите также `find` и `position`.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

[Функция] **rassoc** *item a-list* **&key** *:test :test-not :key*
[Функция] **rassoc-if** *predicate a-list* **&key** *:key*
[Функция] **rassoc-if-not** *predicate a-list* **&key** *:key*

`rassoc` является реверсивной формой для `assoc`. Функция ищет пары, у которых *cdr* элемент удовлетворяет заданному условию. Если *a-list* рассматривается как отображение, то `rassoc` обрабатывает *a-list* как представление инверсного отображения. Например:

(rassoc 'a '((a . b) (b . c) (c . a) (z . a))) ⇒ (c . a)

Выражения

(rassoc *item list* :test *fn*)

и

(find *item list* :test *fn* :key #'cdr)

эквивалентны за исключением, того что `rassoc` игнорирует значения `nil` на месте пар.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

Глава 16

Хеш-таблицы

Хеш-таблица является Lisp'овыми объектом, который может быстро отображать заданный Lisp'овый объект в другой Lisp'овый объект. *Wikipedia более понятно излагает: это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.* Каждая хеш-таблица содержит множество *элементов*, каждый из которых содержит *значение* ассоциированное с *ключом*. Базовые функции взаимодействия с хеш-таблицей могут создавать элементы (пары ключ-значение), удалять элементы и искать значения по заданному ключу. Поиск значения очень быстрый, даже при наличии большого количества элементов, потому что используется хеширование. Это самое важное преимущество хеш-таблиц перед списками свойств.

Хеш-таблица может связывать только одно *значение* с заданным *ключом*. Если вы попробуете добавить второе *значение*, оно заменит предыдущее. К тому же, добавление значения в хеш-таблицу является деструктивной операцией, в этом случае хеш-таблица модифицируется. Ассоциативные списки же, наоборот, могут расширяться недеструктивно.

Хеш-таблицы существуют в трёх видах, различие между ними в том, как сравниваются ключи, с помощью `eq`, `eql` или `equal`. Другими словами, существуют хеш таблицы с ключами, которые используют Lisp'овые *объекты* (`eq` или `eql`) и которые используют *древовидные структуры* (`equal`). FIXME

Хеш-таблицы создаются функцией `make-hash-table`, которая

принимает различные параметры, включая тип хеш-таблицы (по-умолчанию тип `eql`). Для поиска значения по ключу используйте `gethash`. Новые элементы могут быть добавлены с помощью `gethash` внутри `setf`. Для удаления элементов используйте `remhash`. Вот простой пример.

```
(setq a (make-hash-table))
(setf (gethash 'color a) 'brown)
(setf (gethash 'name a) 'fred)
(gethash 'color a) ⇒ brown
(gethash 'name a) ⇒ fred
(gethash 'pointy a) ⇒ nil
```

В этом примере, символы `color` и `name` используются в качестве ключей, а символы `brown` и `fred` в качестве ассоциированных значений. Хеш-таблица содержит две пары, в одной ключ `color` связан с `brown`, а в другой `name` с `fred`.

Ключи необязательно должны быть символами. Они могут быть любыми Lisp'овыми объектами. Значения также могут быть любыми Lisp'овыми объектами.

There is a discrepancy between the preceding description of the size of a hash table and the description of the `:size` argument in the specification below of `make-hash-table`.

X3J13 voted in June 1989 to regard the latter description as definitive: the `:size` argument is approximately the number of entries that can be inserted without having to enlarge the hash table. This definition is certainly more convenient for the user.

16.1 Функции для хеш-таблиц

Данный раздел описывает функции для хеш-таблиц, которые используют *объекты* для ключей и ассоциируют другие объекты с этими.

[Функция] **make-hash-table &key** *:test :size :rehash-size :rehash-threshold*

Эта функция создаёт и возвращает новую хеш-таблицу. Аргумент `:test` определяет как будут сравниваться ключи. Он может быть одним из трёх значений `#'eq`, `#'eql` или `#'equal`, или одним из трёх символов `eq`, `eql` или `equal`. По-умолчанию используется `eql`.

X3J13 voted in January 1989 to add a fourth type of hash table: the value of `#'equalp` and the symbol `equalp` are to be additional valid possibilities for the `:test` argument.

Note that one consequence of the vote to change the rules of floating-point contagion (described in section 12.1) is to require `=`, and therefore also `equalp`, to compare the values of numbers exactly and not approximately, making `equalp` a true equivalence relation on numbers.

Another valuable use of `equalp` hash tables is case-insensitive comparison of keys that are strings.

Аргумент `:size` устанавливает первоначальный размер хеш-таблицы в парах. (Указанный размер может быть округлён до «хорошего» размера, например, до первого следующего простого числа.) Вы можете не сохранять в таблице столько пар, сколько указали. Этот аргумент служит подсказкой для реализации о том, какое примерно число элементов вы будете хранить в хеш-таблице.

X3J13 voted in January 1989 to clarify that the `:size` argument must be a non-negative integer.

X3J13 voted in June 1989 to regard the preceding description of the `:size` argument as definitive: it is approximately the number of entries that can be inserted without having to enlarge the hash table.

Аргумент `:rehash-size` указывает, на сколько увеличить размер хеш-таблицы, когда она по размерам достигнет пределов. Если это целое число, оно должно быть больше нуля, и будет означать абсолютное приращение. Если это число с плавающей точкой, оно должно быть больше 1, и будет означать относительное приращение к предыдущему размеру. Значение по-умолчанию зависит от реализации.

Аргумент `:rehash-threshold` указывает, насколько должна наполниться хеш-таблица, прежде чем она будет увеличена. Он должен быть числом типа `real` между `0` и `1`, включительно. Он показывает максимальный уровень заполнения хеш-таблицы. Значение по-умолчанию зависит от реализации.

Пример использования `make-hash-table`:

```
(make-hash-table :rehash-size 1.5
                 :size (* number-of-widgets 43))
```

[Функция] **hash-table-p** *object*

`hash-table-p` возвращает истину, если аргумент является хеш-таблицей. Иначе возвращает ложь.

(hash-table-p x) ≡ (typep x 'hash-table)

[Функция] **gethash** *key hash-table* **&optional** *default*

`gethash` ищет элемент в хеш-*hash-table*, чей ключ равен *key* и возвращает связанное значение. Если элемент не был найден, то возвращается значение аргумента *default*, который по-умолчанию равен `nil`.

`gethash` возвращает два значение. Второе значение является предикатом, и истинно, если значение было найдено, и ложно если нет.

`setf` может использоваться вместе с `gethash` для создания в хеш-таблице новых элементов. Если элемент с заданным ключом *key* уже существует, он будет удалён перед добавлением. В этом контексте может использоваться аргумент *default*, он игнорируется `setf`, но может быть полезным в таких макросах как `incf`, которые связаны с `setf`:

```
(incf (gethash a-key table 0))
```

обозначает то же, что и

```
(setf (gethash a-key table 0)
      (+ (gethash a-key table 0) 1))
```

что можно преобразовать в

```
(setf (gethash a-key table)
      (+ (gethash a-key table 0) 1))
```

[Функция] **remhash** *key hash-table*

`remhash` удаляет любой элемент в хеш-таблице *hash-table* ключ, которого равен параметру *key*. Возвращает истину, если элемент был удалён, и ложь, если элемента уже не существовало.

[Функция] **maphash** *function hash-table*

Для каждого элемента в хеш-таблице *hash-table*, `maphash` вызывает функцию *function* с двумя аргументами: ключ элемента и значение элемента. `maphash` возвращает `nil`. Если во время выполнения `maphash` в хеш-таблице добавлялись или удалялись ключи, то результат непредсказуем, но есть исключение: если функция *function* вызывает `remhash` для удаления элемента, который в эту функцию и был передан, или устанавливает новое значение с помощью `setf` этому элементу, то эти операции будут выполнены правильно. Например:

```
;;; Изменение каждого элемента в MY-HASH-TABLE, с заменой на
;;; квадратный корень. Элементы с отрицательными значения удаляются.
(maphash #'(lambda (key val)
             (if (minusp val)
                 (remhash key my-hash-table)
                 (setf (gethash key my-hash-table) (sqrt val))))
         my-hash-table)
```

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

[Функция] **clrhash** *hash-table*

Функция удаляет все элемент из хеш-таблицы *hash-table* и возвращает эту хеш-таблицу.

[Функция] **hash-table-count** *hash-table*

Функция возвращает количество элементов в хеш-таблице *hash-table*. Когда хеш-таблица только были сделана, или только что очищена, то количество элементов равно нулю.

[Макрос] **with-hash-table-iterator** (mname hash-table) {form}*

X3J13 voted in January 1989 to add the macro `with-hash-table-iterator`.

The name *mname* is bound and defined as if by `macrolet`, with the body *form*s as its lexical scope, to be a "generator macro" such that successive invocations `(`*mname*`)` will return entries, one by one, from the hash table that is the value of the expression *hash-table* (which is evaluated exactly once).

At each invocation of the generator macro, there are two possibilities. If there is yet another unprocessed entry in the hash table, then three values are returned: `t`, the key of the hash table entry, and the associated value of the hash table entry. On the other hand, if there are no more unprocessed entries in the hash table, then one value is returned: `nil`.

The implicit interior state of the iteration over the hash table entries has dynamic extent. While the name *mname* has lexical scope, it is an error to invoke the generator macro once the `with-hash-table-iterator` form has been exited.

Invocations of `with-hash-table-iterator` and related macros may be nested, and the generator macro of an outer invocation may be called from within an inner invocation (assuming that its name is visible or otherwise made available).

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

Обоснование: This facility is a bit more flexible than `maphash`. It makes possible a portable and efficient implementation of `loop` clauses for iterating over hash tables (see chapter 25).

```
(setq turtles (make-hash-table :size 9 :test 'eq))
(setf (gethash 'howard-kaylan turtles) '(musician lead-singer))
(setf (gethash 'john-barbata turtles) '(musician drummer))
(setf (gethash 'leonardo turtles) '(ninja leader blue))
(setf (gethash 'donatello turtles) '(ninja machines purple))
(setf (gethash 'al-nichol turtles) '(musician guitarist))
(setf (gethash 'mark-volman turtles) '(musician great-hair))
(setf (gethash 'raphael turtles) '(ninja cool rude red))
(setf (gethash 'michaelangelo turtles) '(ninja party-dude orange))
(setf (gethash 'jim-pons turtles) '(musician bassist))
```

```
(with-hash-table-iterator (get-turtle turtles)
  (labels ((try (got-one &optional key value)
             (when got-one          ;Remember, keys may show up in any order
               (when (eq (first value) 'ninja)
                 (format t "~%~:(~A~): ~{~A~^, ~}"
                         key (rest value)))
               (multiple-value-call #'try (get-turtle)))))
    (multiple-value-call #'try (get-turtle))))     ;Prints 4 lines
Michaelangelo: PARTY-DUDE, ORANGE
Leonardo: LEADER, BLUE
Raphael: COOL, RUDE, RED
Donatello: MACHINES, PURPLE
  ⇒ nil
```

[Функция] **hash-table-rehash-size** *hash-table*
[Функция] **hash-table-rehash-threshold** *hash-table*
[Функция] **hash-table-size** *hash-table*
[Функция] **hash-table-test** *hash-table*

Добавлены функции, которые возвращают значения используемые при вызове `make-hash-table`.

`hash-table-rehash-size` возвращает размер приращения хеш-таблицы.

`hash-table-rehash-threshold` возвращает текущий уровень заполнения.

`hash-table-size` возвращает рекомендуемый размер хеш-таблицы.

`hash-table-test` возвращает функцию сравнения используемую для ключей. Если данная функция одна из стандартных, то результат всегда является символом, даже если при создании хеш-таблицы было указано иное. Например:

```
(hash-table-test (make-hash-table :test #'equal)) ⇒ equal
```

Реализации, которые расширяют `make-hash-table` дополнительными функциями сравнения для аргумента `:test`, могут определять, как будет

возвращено значение из `hash-table-test` для этих дополнительных функций.

16.2 Функция хеширования

Функция `sxhash` является удобным инструментом для пользователя, нуждающегося в создании более сложных хешированных структур данных, чем предоставляются объектами `hash-table`.

[Функция] **sxhash** *object*

`sxhash` вычисляет хеш для объекта и возвращает этот хеш, как неотрицательное число fixnum. Свойство `sxhash` заключается в том, что `(equal` *x y*`)` подразумевает `(= (sxhash` *x*`) (sxhash` *y*`))`.

Механизм вычисления хеша зависит от реализации, но независим от запущенного экземпляра. Например, вычисленные `sxhash` хеши могут быть записаны в файлы, и без потери информации прочитаны из них в рамках одной реализации.

Глава 17

Массивы

Массив является объектом с элементами, расположенными в соответствие с прямолинейной координатной системой. В целом, это можно назвать построчным хранением объектов многомерных таблиц. В теории, массив в Common Lisp'е может иметь любое количество измерений, включая ноль. (Нульмерный массив измерений имеет только один элемент. FIXME) На практике, реализация может ограничивать количество измерений, но должна как минимум поддерживать семь. Каждое измерение является неотрицательным целым. Если любое из измерений равно нулю, то массив не имеет элементов.

Массив может быть *общим*. Это означает, что элемент может быть любым Lisp объектом. Массив также может быть *специализированным*. Это означает, что элемент ограничен некоторым типом.

17.1 Создание массива

Не волнуйтесь о большом количестве опций для функции `make-array`. Все что действительно необходимо, это список размерностей. Остальные же опции служат для относительно эзотерических программ.

[Функция] **make-array** *dimensions* **&key** *:element-type :initial-element :initial-contents :adjustable :fill-pointer :displaced-to :displaced-index-offset*

Это базовая функция для создания массивов. Аргумент *dimensions* должен быть списком неотрицательных целых чисел,

которые являются измерениями массива. Длина списка является размерностью массива. Каждое измерение должно быть меньше чем `array-dimension-limit`, и произведение всех измерений должно быть меньше `array-total-size-limit`. Следует отметить, что если *dimensions* `nil`, тогда создаётся массив с нулевым количеством измерений. По соглашению, при создании одномерного массива, размер измерений может быть указан как одиночное целое число, а не список целых чисел.

Реализация Common Lisp'а может иметь ограничение на ранг массива, но это ограничение должно быть меньше 7. Таким образом, любая Common Lisp программа может использовать массивы с рангом меньшим или равным 7.

Для `make-array` существуют следующие именованные параметры:

`:element-type` Этот аргумент должен быть именем типа элементов массива. Массив создаётся с наиболее близким типом к указанному, и не может содержать объекты отличные от этого типа. Тип `t` указывает на создание общего массива, элементы которого могут быть любыми объектами. `t` является значением по-умолчанию.

X3J13 voted in January 1989 to change `typep` and `subtypep` so that the specialized `array` type specifier means the same thing for discrimination purposes as for declaration purposes: it encompasses those arrays that can result by specifying *element-type* as the element type to the function `make-array`. Therefore we may say that if *type* is the `:element-type` argument, then the result will be an array of type `(array` *`type`*`)`; put another way, for any type A,

```
(typep (make-array ... :element-type 'A ...)
       '(array A)))
```

is always true. See `upgraded-array-element-type`.

`:initial-element` Этот аргумент может использоваться для инициализации каждого элемента массива. Значение должно принадлежать типу указанному в аргументе `:element-type`. Если `:initial-element` опущен, тогда значение по-умолчанию для элементов массива не определено (если только не используется

`:initial-contents` или `:displaced-to`). Опция `:initial-element` может не использоваться, когда указана опция `:initial-contents` или `:displaced-to`.

`:initial-contents` Этот аргумент может использоваться для инициализации содержимого массива. Значением параметра является структура вложенных последовательностей. Если массив с нулевым количеством измерений, тогда значение задаёт одиночный элемент. В противном случае, значение должно быть последовательностью с длинной, равной размеру первого измерения. Каждый вложенный элемент также должен быть последовательностью, размером равным следующему измерению массива. Например:

```
(make-array '(4 2 3)
            :initial-contents
            '(((a b c) (1 2 3))
              ((d e f) (3 1 2))
              ((g h i) (2 3 1))
              ((j k l) (0 0 0))))
```

Количество уровней в структуре должно равняться рангу массива. Каждый лист вложенной структуры должен иметь тип, указанный в опции `:type`. Если опция `:initial-contents` опущена, первоначальное значение элементов массива не определено (если только не используется опция `:initial-element` или `:displaced-to`). Опция `:initial-contents` может не использоваться, когда указана опция `:initial-element` или `:displaced-to`.

`:adjustable` Если данный аргумент указан и не является `nil`, тогда размер массива после создания может быть изменён. По-умолчанию параметр равен `nil`.

X3J13 voted in June 1989 to clarify that if this argument is non-`nil` then the predicate `adjustable-array-p` will necessarily be true when applied to the resulting array; but if this argument is `nil` (or omitted) then the resulting array may or may not be adjustable, depending on

the implementation, and therefore `adjustable-array-p` may be correspondingly true or false of the resulting array. Common Lisp provides no portable way to create a non-adjustable array, that is, an array for which `adjustable-array-p` is guaranteed to be false.

`:fill-pointer` Данный аргумент указывает, что массив должен иметь указатель заполнения. Если эта опция указана и не является `nil`, тогда массив должен быть одномерным. Значение используется для инициализации указателя заполнения массива. Если указано значение `t`, тогда используется длина массива. Иначе, значение должно быть целым числом между нулём (включительно) и длинной массива (включительно)ю Аргумент по-умолчанию равен `nil`.

`:displaced-to` Если данный аргумент указан и не является `nil`, то он указывает на то, что массив будет *соединён* с другим массивом. Значение аргумента должно быть уже созданным массивом. `make-array` создаст *косвенный* или *разделяемый* массив, который имеет общее содержимое с указанным массивом. В таком случае опция `:displaced-index-offset` позволяет указать смещение общей части массивов. Если массив, переданный в аргумент `:displaced-to`, не имеет такого же типа элементов (`:element-type`), как и создаваемый массив, то генерируется ошибка. Опция `:displaced-to` не может использоваться вместе с опцией `:initial-element` или `:initial-contents`. По-умолчанию аргумент равен `nil`.

`:displaced-index-offset` Данный аргумент может использоваться только в сочетании с опцией `:displaced-to`. Она должна быть неотрицательным целым (по-умолчанию равна нулю). Она означает смещение создаваемого *разделяемого* массива относительно исходного.

Если при создании массива B массив A передан в качестве аргумента `:displaced-to` в функцию `make-array`, тогда массив B называется *соединённым* с массивом A. Полное количество элементов массива, называемое *полный размер* массива, вычисляется как произведение размеров всех измерений (смотрите `array-total-size`). Необходимо, чтобы полный размер массива A

был не меньше чем сумма *полного размера* массива B и смещения n, указанного в аргументе `:displaced-index-offset`. Смысл соединения массивов в том, что массив B не имеет своих элементов и операции над ним приводят в действительности к операциям над массивом A. При отображении элементов массивов, последние рассматриваются как одномерные, с построчным порядком размещения элементов. Так, элемент с индексом k массива B отображается в элемент с индексом $k+n$ массива A.

Если `make-array` вызывается без указания аргументов `:adjustable`, `:fill-pointer` и `:displaced-to`, или указания их в `nil`, тогда получаемый массив гарантированного будет *простым* (смотрите раздел 2.5).

X3J13 voted in June 1989 to clarify that if one or more of the `:adjustable`, `:fill-pointer`, and `:displaced-to` arguments is true, then whether the resulting array is simple is unspecified.

Вот несколько примеров использования `make-array`:

```
;;; Создание одномерного массива с пятью элементами.
(make-array 5)

;;; Создание двумерного массива, 3 на 4, с четырёхбитными элементами.
(make-array '(3 4) :element-type '(mod 16))

;;; Создание массива чисел с плавающей точкой.
(make-array 5 :element-type 'single-float))

;;; Создание соединённого массива.
(setq a (make-array '(4 3)))
(setq b (make-array 8 :displaced-to a
                      :displaced-index-offset 2))
```

;;; В таком случае:
(aref b 0) ≡ (aref a 0 2)
(aref b 1) ≡ (aref a 1 0)
(aref b 2) ≡ (aref a 1 1)
(aref b 3) ≡ (aref a 1 2)
(aref b 4) ≡ (aref a 2 0)
(aref b 5) ≡ (aref a 2 1)
(aref b 6) ≡ (aref a 2 2)
(aref b 7) ≡ (aref a 3 0)

Последний пример связи между массивами зависит от того факта, что элементы массивов хранятся построчно. Другими словами, индексы элментов массива упорядочены лексикографически.

[Константа] **array-rank-limit**

Значение `array-rank-limit` является положительным целым, которое обозначает (невключительно) наивысший возможный ранг массива. Это значение зависит от реализации, но не может быть меньше 8. Таком образом каждая реализация Common Lisp'а поддерживает массивы, ранг которых лежит между 0 и 7 (включительно). (Разработчикам предлагается сделать это значение большим настолько, на сколько это возможно без ущерба в производительности.)

[Константа] **array-dimension-limit**

Значение `array-dimension-limit` является положительным целым, которое обозначает (невключительно) наивысший возможный размер измерения для массива. Это значение зависит от реализации, но не может быть меньше 1024. (Разработчикам предлагается сделать это значение большим настолько, на сколько это возможно без ущерба в производительности.)

X3J13 voted in January 1989 to specify that the value of `array-dimension-limit` must be of type `fixnum`. This in turn implies that all valid array indices will be fixnums.

[Константа] **array-total-size-limit**

Значение `array-total-size-limit` является положительным целым, которое обозначает (невключительно) наивысшее возможное количество элементов в массиве. Это значение зависит от реализации, но не может быть меньше 1024. (Разработчикам предлагается сделать это значение большим настолько, на сколько это возможно без ущерба в производительности.)

На практике предельный размер массива может зависеть от типа элемента массива, в таком случае `array-total-size-limit` будет наименьшим из ряда этих размеров.

[Функция] **vector** **&rest** *objects*

Функция `vector` служит для создания простых базовых векторов с заданным содержимым. Она является аналогом для функции `list`.

(vector a_1 a_2 ... a_n)
≡ (make-array (list n) :element-type t
:initial-contents (list a_1 a_2 ... a_n))

17.2 Доступ к массиву

Для доступа к элементам массива обычно используется функция `aref`. В отдельных случаях может быть более эффективно использование других функций, таких как `svref`, `char` и `bit`.

[Функция] **aref** *array* **&rest** *subscripts*

Данная функция возвращает элемент массива *array*, который задан индексами *subscripts*. Количество индексов должно совпадать с рангом массива, и каждый индекс должен быть неотрицательным целым числом меньшим чем соответствующий размер измерения.

`aref` отличается от остальных функций тем, что полностью игнорирует указатели заполнения. `aref` может получать доступ к любому элементу массива, вне зависимости является он активным или

нет. Однако общая функция для последовательностей `elt` учитывает указатель заполнения. Доступ к элементам за указателем заполнение с помощью `elt` является ошибкой.

Для изменения элемента массива может использоваться `setf` в связке с `aref`.

В некоторых случаях, удобно писать код, который получает элемент из массива, например `a`, используя список индексов, например `z`. Это легко сделать используя `apply`:

```
(apply #'aref a z)
```

(Длина списка, конечно же, должна быть равна рангу массива.) Эта конструкция может использоваться с `setf` для изменения элемента массива на, например, `w`:

```
(setf (apply #'aref a z) w)
```

[Функция] **svref** *simple-vector index*

Первый аргумент должен быть простым базовым вектором, то есть объектом типа `simple-vector`. В результате возвращается элемент вектора *simple-vector* с индексом *index*.

index должен быть неотрицательным целым меньшим чем длина вектора.

Для изменения элемента вектора может использоваться `setf` в связке с `svref`.

`svref` идентична `aref` за исключением того, что требует, чтобы первый аргумент был простым вектором. В некоторых реализациях Common Lisp'a, `svref` может быть быстрее чем `aref` в тех случаях, где она применима. Смотрите также `schar` и `sbit`

17.3 Информация о массиве

Следующие функции извлекают интересную информацию, и это не элементы массива.

[Функция] **array-element-type** *array*

`array-element-type` возвращает спецификатор типа для множества объектов, которые могут быть сохранены в массиве *array*. Это множество может быть больше чем то, которое запрашивалось в функции `make-array`. Например, результат

```
(array-element-type (make-array 5 :element-type '(mod 5)))
```

может быть `(mod 5)`, `(mod 8)`, `fixnum`, `t` или любой другой тип, для которого `(mod 5)` является подтипом. Смотрите `subtypep`.

[Функция] **array-rank** *array*

Эта функция возвращает количество измерений (осей) массива *array*. Результат будет неотрицательным целым. Смотрите `array-rank-limit`.

[Функция] **array-dimension** *array axis-number*

Данная функция возвращает размер измерения *axis-number* массива *array*. *array* может быть любым видом массива, и *axis-number* должен быть неотрицательным целым меньшим чем ранг массива *array*. Если *array* является вектором с указателем заполнения, `array-dimension` возвращает общий размер вектора, включая неактивные элементы, а не размер ограниченный указателем заполнения. (Функция `length` будет возвращать размер ограниченный указателем заполнения.)

[Функция] **array-dimensions** *array*

`array-dimensions` возвращает список, элементы которого являются размерами измерений массива *array*.

[Функция] **array-total-size** *array*

`array-total-size` возвращает общее количество элементов массива *array*, которое вычислено как произведение размеров всех измерений.

(array-total-size x)
  $\equiv$ (apply #'* (array-dimensions x))
  $\equiv$ (reduce #'* (array-dimensions x))

Следует отметить, что общий размер нульмерного (FIXME) массива равен `1`. Общий размер одномерного массива вычисляется без учёта указателя заполнения.

[Функция] **array-in-bounds-p** *array* **&rest** *subscripts*

Данный предикат проверяет, являются ли индексы *subscripts* для массива *array* корректными. Если они корректны, предикат истинен, иначе ложен. *subscripts* должен быть целыми числами. Количество индексов *subscripts* должно равняться рангу массива. Как и `aref`, `array-in-bounds-p` игнорирует указатели заполнения.

[Функция] **array-row-major-index** *array* **&rest** *subscripts*

Данная функция принимает массив и корректные для него индексы и возвращает одиночное неотрицательное целое значение меньшее чем общий размер массива, которое идентифицирует элемент, полагаясь на построчный порядок хранения элементов. Количество указанных индексов *subscripts* должно равняться рангу массива. Каждый индекс должен быть неотрицательным целым числом меньшим чем соответствующий размер измерения. Как и `aref`, `array-row-major-index` игнорирует указатели заполнения.

Возможно определение `array-row-major-index`, без проверки на ошибки, может выглядеть так:

```
(defun array-row-major-index (a &rest subscripts)
  (apply #'+ (maplist #'(lambda (x y)
                          (* (car x) (apply #'* (cdr y))))
                      subscripts
                      (array-dimensions a))))
```

Для одномерного массива, результат `array-row-major-index` всегда равен переданному индексу.

[Функция] **row-major-aref** *array index*

Данная функция позволяет получить доступ к элементу, как если бы массив был одномерный. Аргумент *index* должен быть неотрицательным

целым меньшим чем общий размер массива *array*. Данная функция индексирует массив, как если бы он был одномерный с построчным порядком. Эту функцию можно понять в терминах `aref`:

```
(row-major-aref array index) ≡
  (aref (make-array (array-total-size array))
                :displaced-to array
                :element-type (array-element-type array))
        index)
```

Другими словами, можно обработать массив как одномерный с помощью создания нового одномерного массива, который *соединён* с исходным, и получить доступ к новому массиву. И наоборот, `aref` может быть описана в терминах `row-major-aref`:

```
(aref array i0 i1 ... in−1) ≡
  (row-major-aref array
                (array-row-major-index array i0 i1 ... in−1)
```

Как и `aref`, `row-major-aref` полностью игнорирует указатели заполнения. Для изменения элемента массива, можно комбинировать вызов `row-major-aref` с формой `setf`.

Эта операция облегчает написание кода, который обрабатывает массивы различных рангов. Предположим, что необходимо обнулить содержимое массива `tennis-scores`. Можно решить это так:

```
(fill (make-array (array-total-size tennis-scores)
                :element-type (array-element-type tennis-scores)
                :displaced-to tennis-scores)
      0)
```

К сожалению, так как `fill` не может принимать многомерные массивы, в данном примере создаётся *соединённый* массив, что является лишней операцией. Другим способом является отдельная обработка каждого измерения многомерного массива:

```
(ecase (array-rank tennis-scores)
  (0 (setf (aref tennis-scores) 0))
  (1 (dotimes (i0 (array-dimension tennis-scores 0))
       (setf (aref tennis-scores i0) 0)))
  (2 (dotimes (i0 (array-dimension tennis-scores 0))
       (dotimes (i1 (array-dimension tennis-scores 1))
         (setf (aref tennis-scores i0 i1) 0))))
  ...
  (7 (dotimes (i0 (array-dimension tennis-scores 0))
       (dotimes (i1 (array-dimension tennis-scores 1))
         (dotimes (i2 (array-dimension tennis-scores 1))
           (dotimes (i3 (array-dimension tennis-scores 1))
             (dotimes (i4 (array-dimension tennis-scores 1))
               (dotimes (i5 (array-dimension tennis-scores 1))
                 (dotimes (i6 (array-dimension tennis-scores 1))
                   (setf (aref tennis-scores i0 i1 i2 i3 i4 i5 i6)
                         0)))))))))
  )
```

От такого кода быстро приходит усталость. Кроме того, данный подход не желателен, так как некоторые реализации Common Lisp'a будут фактически поддерживать не более 7 измерений. Рекурсивно вложенные циклы справляются с задачей лучше, но код всё ещё выглядит как лапша:

```
(labels
  ((grok-any-rank (&rest indices)
     (let ((d (- (array-rank tennis-scores) (length indices)))
       (if (= d 0)
           (setf (apply #'row-major-aref indices) 0)
           (dotimes (i (array-dimension tennis-scores (- d 1)))
             (apply #'grok-any-rank i indices))))))
  (grok-any-rank))
```

Является ли этот код эффективным зависит от многих параметров реализации, таких как способ обработки `&rest` аргументов и компиляции

`apply` вызовов. Только посмотрите как просто использовать для задачи `row-major-aref`!

```
(dotimes (i (array-total-size tennis-scores))
  (setf (row-major-aref tennis-scores i) 0))
```

Нет сомнения, что этот код, слаще любых медовых сот.

[Функция] **adjustable-array-p** *array*

Если аргумент, который должен быть массивом, может быть расширен, данный предикат истинен, иначе ложен.

X3J13 voted in June 1989 to clarify that `adjustable-array-p` is true of an array if and only if `adjust-array`, when applied to that array, will return the same array, that is, an array `eq` to the original array. If the `:adjustable` argument to `make-array` is non-`nil` when an array is created, then `adjustable-array-p` must be true of that array. If an array is created with the `:adjustable` argument `nil` (or omitted), then `adjustable-array-p` may be true or false of that array, depending on the implementation. X3J13 further voted to *define* the terminology "adjustable array" to mean precisely "an array of which `adjustable-array-p` is true." See `make-array` and `adjust-array`.

[Функция] **array-displacement** *array*

функция возвращает два значение. Первое значение является массивом соединенным с данным, и второе значение обозначает смещение соединения. Если массив не был соединен ни с одним массивом возвращаются значения `nil` и `0`.

17.4 Функции для битовых массивов

Функции описанные в данном разделе работают только с массивами битов, то есть, со специализированными массивами, элементы которых могут принимать значения только либо `0`, либо `1`.

[Функция] **bit** *bit-array* **&rest** *subscripts*
[Функция] **sbit** *simple-bit-array* **&rest** *subscripts*

Функция `bit` похожа на `aref`, но принимает только массив битов, то есть массив типа `(array bit)`. Результатом всегда является `0` или `1`. `sbit` похожа на `bit`, но дополнительно требует, чтобы первый аргумент был *простым* массивом (смотрите раздел 2.5). Следует отметить, что `bit` и `sbit`, в отличие от `char` и `schar`, могут принимать массив любого ранга.

Для замены элемента массива может использоваться `setf` в связке с `bit` или `sbit`.

`bit` и `sbit` идентичны `aref` за исключением того, что принимают только специализированные массивы. В некоторых реализациях Common Lisp'а `bit` и `sbit` могут быть быстрее чем `aref` в тех случаях, где они применимы.

[Функция] **bit-and** *bit-array1 bit-array2* **&optional** *result-bit-array*
[Функция] **bit-ior** *bit-array1 bit-array2* **&optional** *result-bit-array*
[Функция] **bit-xor** *bit-array1 bit-array2* **&optional** *result-bit-array*
[Функция] **bit-eqv** *bit-array1 bit-array2* **&optional** *result-bit-array*
[Функция] **bit-nand** *bit-array1 bit-array2* **&optional** *result-bit-array*
[Функция] **bit-nor** *bit-array1 bit-array2* **&optional** *result-bit-array*
[Функция] **bit-andc1** *bit-array1 bit-array2* **&optional** *result-bit-array*
[Функция] **bit-andc2** *bit-array1 bit-array2* **&optional** *result-bit-array*
[Функция] **bit-orc1** *bit-array1 bit-array2* **&optional** *result-bit-array*
[Функция] **bit-orc2** *bit-array1 bit-array2* **&optional** *result-bit-array*

Эти функции выполняют побитовые логические операции над битовыми массивами. Все аргументы для этих функций должны быть битовыми массивами одинакового ранга и измерений. Результат является битовым массивом с такими же рангом и измерениями, что и аргументами. Каждый бит получается применением соответствующей операции к битам исходных массивов.

Если третий аргумент опущен или является `nil`, создаётся новый массив, который будет содержать результат. Если третий аргумент это битовый массив, то результат помещается в этот массив. Если третий аргумент является `t`, тогда для третьего аргумента используется первый массив. Таким образом результат помещается в массив переданный первым аргументом.

Следующая таблица отображает результаты применения битовых операций.

argument1	0	0	1	1	
argument2	0	1	0	1	*Имя операции*
bit-and	0	0	0	1	и
bit-ior	0	1	1	1	или
bit-xor	0	1	1	0	исключающее или
bit-eqv	1	0	0	1	равенство (исключающее не-или)
bit-nand	1	1	1	0	не-и
bit-nor	1	0	0	0	не-или
bit-andc1	0	1	0	0	не-*argument1* и *argument2*
bit-andc2	0	0	1	0	*argument1* и не-*argument2*
bit-orc1	1	1	0	1	не-*argument1* или *argument2*
bit-orc2	1	0	1	1	*argument1* или не-*argument2*

Например:

(bit-and #*1100 #*1010) ⇒ #*1000
(bit-xor #*1100 #*1010) ⇒ #*0110
(bit-andc1 #*1100 #*1010) ⇒ #*0100

Смотрите также `logand` и связанные с ней функции.

[Функция] **bit-not** *bit-array* **&optional** *result-bit-array*

Первый аргумент должен быть массивом битов. Результатом является битовый массив с таким же рангом и измерениями и инвертированными битами. Смотрите также `lognot`.

Если второй аргумент опущен или равен `nil`, для результата создаётся новый массив. Если второй аргумент является битовым массивом, результат помещается в него. Если второй аргумент является `t`, то для результата используется массив из первого аргумента. То есть результат помещается в исходный массив.

17.5 Указатели заполнения

Common Lisp предоставляет несколько функций для управления *указателями заполнения*. Они позволяют последовательно заполнять содержимое вектора. А если точнее, они позволяют эффективно менять длину вектора. Например, строка с указателем заполнения имеет большинство характеристик строки с переменной длиной из PL/I.

Указатель заполнения является неотрицательным целым числом не большим чем общее количество элементов в векторе (полученное от `array-dimension`). Указатель заполнения указывает на «активные» или «заполненные» элементы вектора. Указатель заполнение отображает «активную длину» вектора. Все элементы вектора, индекс которых меньше чем указатель заполнения, являются активными. Остальные элементы являются неактивными. Почти все функции, которые оперируют содержимым вектора, будут оперировать только активными элементами. Исключением является `aref`, которая может получать доступ ко всем элементам массива, активным и неактивным. Следует отметить, что элементы вектора из неактивной части тем не менее являются частью вектора.

Заметка для реализации: An implication of this rule is that vector elements outside the active region may not be garbage-collected.

Указатели заполнения могут иметь только вектора (одномерные массивы). Многомерные массивы не могут иметь указатели заполнения. (Следует отметить, однако, что можно создать многомерный массив *соединённый* с вектором, у которого есть указатель заполнения.)

[Функция] **array-has-fill-pointer-p** *array*

Аргумент должен быть массивом. `array-has-fill-pointer-p` возвращает `t`, если массив умеет указатель заполнения, иначе возвращает `nil`. Следует отметить, что `array-has-fill-pointer-p` всегда возвращает `nil`, если массив *array* не одномерный.

[Функция] **fill-pointer** *vector*

Данная функция возвращает указатель заполнения вектора *vector*. Если вектор *vector* не имеет указателя заполнения, генерируется ошибка.

Для изменения указателя заполнения вектора может использоваться функция `setf` в связке с `fill-pointer`. Указатель заполнения вектора должен быть всегда целым числом между нулём и размером вектора (включительно).

[Функция] **vector-push** *new-element vector*

Аргумент *vector* должен быть одномерным массивом, имеющим указатель заполнения, и *new-element* может быть любым объектом. `vector-push` пытается сохранить *new-element* в элемент вектора, на который ссылается указатель заполнения, и увеличить этот указатель на 1. Если указатель заполнения не определяет элемент вектора (например, когда он становится слишком большим), то `vector-push` возвращает `nil`. В противном случае, если вставка нового элемента произошла, `vector-push` возвращает *предыдущее* значение указателя. Таким образом, `vector-push` является индексом вставленного элемента.

Можно сравнить `vector-push`, которая является функцией, с `push`, который являет макросом, который принимает *место* подходящее для `setf`. Вектор с указателем заполнения содержит такое *место* в слоте `fill-pointer`. FIXME

[Функция] **vector-push-extend** *new-element vector* **&optional** *extension*

`vector-push-extend` похожа на `vector-push` за исключением того, что если указатель заполнения стал слишком большим, длина вектора увеличивается (с помощью `adjust-array`), и новый элемент помещается в вектор. Однако, если вектор не расширяемый, тогда `vector-push-extend` сигнализирует ошибку.

X3J13 voted in June 1989 to clarify that `vector-push-extend` regards an array as not adjustable if and only if `adjustable-array-p` is false of that array.

Необязательный аргумент *extension*, который должен быть положительным целым, является минимальным количеством элементов, добавляемых в вектор, если последний должен быть расширен. По-умолчанию содержит значение зависимое от реализации.

[Функция] **vector-pop** *vector*

Аргумент *vector* должен быть одномерным массивом, который

имеет указатель заполнения. Если указатель заполнения является нулём, `vector-pop` сигнализирует ошибку. В противном случае, указатель заполнения уменьшается на 1, и в качестве значения функции возвращается обозначенный указателем элемент вектора.

17.6 Изменение измерений массива

Следующие функции могут использоваться для изменения размера или формы массива. Их опции почти совпадают с опциями функции `make-array`.

[Функция] **adjust-array** *array new-dimensions* **&key** *:element-type :initial-element :initial-contents :fill-pointer :displaced-to :displaced-index-offset*

`adjust-array` принимает массив и те же аргументы, что и для `make-array`. Количество измерений, указанных в *new-dimensions* должно равняться рангу массива *array*.

`adjust-array` возвращает массив такого же типа и ранга, что и массив *array*, но с другим измерениями *new-dimensions*. Фактически, аргумент *array* модифицируется в соответствие с новыми указаниями. Но это может быть достигнуто двумя путями: модификацией массива *array* или созданием нового массива и модификацией аргумента *array* для *соединения* его с новым массивом.

В простейшем случае, можно указать только измерения *new-dimensions* и, возможно, аргумент `:initial-element`. Не пустые элементы массива *array* вставляются в новый массив. Элементы нового массива, которые не были заполнены значениями из старого массива, получают значение из `:initial-element`. Если аргумент не был указан, то первоначальное значение элементов не определено.

Если указан `:element-type`, тогда массив *array* должен был быть создан с указанием такого же типа. В противном случае сигнализируется ошибка. Указание `:element-type` в `adjust-array` служит только для проверки на существование ошибки несовпадения типов.

Если указаны `:initial-contents` или `:displaced-to`, тогда они обрабатываются как для `make-array`. В таком случае,

Если указан `:fill-pointer`, тогда указатель заполнения массива *array* принимает указанное значение. Если массив *array* не содержал указателя заполнения сигнализируется ошибка.

X3J13 voted in June 1988 to clarify the treatment of the `:fill-pointer` argument as follows.

If the `:fill-pointer` argument is not supplied, then the fill pointer of the *array* is left alone. It is an error to try to adjust the *array* to a total size that is smaller than its fill pointer.

If the `:fill-pointer` argument is supplied, then its value must be either an integer, `t`, or `nil`. If it is an integer, then it is the new value for the fill pointer; it must be non-negative and no greater than the new size to which the *array* is being adjusted. If it is `t`, then the fill pointer is set equal to the new size for the *array*. If it is `nil`, then the fill pointer is left alone; it is as if the argument had not been supplied. Again, it is an error to try to adjust the *array* to a total size that is smaller than its fill pointer.

An error is signaled if a non-`nil` `:fill-pointer` value is supplied and the *array* to be adjusted does not already have a fill pointer.

This extended treatment of the `:fill-pointer` argument to `adjust-array` is consistent with the previously existing treatment of the `:fill-pointer` argument to `make-array`.

`adjust-array` может, в зависимости от реализации и аргументов, просто изменить исходный массив или создать и вернуть новый. В последнем случае исходный массив изменяется, а именно *соединяется* с новым массивом, и имеет новые измерения.

X3J13 voted in January 1989 to allow `adjust-array` to be applied to any array. If `adjust-array` is applied to an array that was originally created with `:adjustable` true, the array returned is `eq` to its first argument. It is not specified whether `adjust-array` returns an array `eq` to its first argument for any other arrays. If the array returned by `adjust-array` is not `eq` to its first argument, the original array is unchanged and does not share storage with the new array.

Under this new definition, it is wise to treat `adjust-array` in the same manner as `delete` and `nconc`: one should carefully retain the returned value, for example by writing

```
(setq my-array (adjust-array my-array ...))
```

rather than relying solely on a side effect.

Если `adjust-array` применяется к массиву *array*, который соединён с другим массивом *x*, тогда ни массив *array*, ни возвращённый результат не будет *соединены* с *x*, если только такое соединения не задано явно в вызове `adjust-array`.

Например, предположим что массив 4-на-4 `m` выглядит так:

```
#2A( ( alpha    beta     gamma    delta )
   ( epsilon  zeta     eta      theta )
   ( iota     kappa    lambda   mu    )
   ( nu       xi       omicron  pi    ) )
```

Тогда результат выражения

```
(adjust-array m '(3 5) :initial-element 'baz)
```

является массивом 3-на-5 с содержимым

```
#2A( ( alpha    beta     gamma    delta    baz )
   ( epsilon  zeta     eta      theta    baz )
   ( iota     kappa    lambda   mu       baz ) )
```

Следует отметить, что если массив `a` создаётся *соединённым* с массивом `b` и затем массив `b` передаётся в `adjust-array`, то массив `a` все ещё будет *соединён* с массивом `b`. При этом должны быть приняты во внимание правила *соединения* массивов и построчный порядок следования элементов.

X3J13 voted in June 1988 to clarify the interaction of `adjust-array` with array displacement.

Suppose that an array *A* is to be adjusted. There are four cases according to whether or not *A* was displaced before adjustment and whether or not the result is displaced after adjustment.

- Suppose *A* is not displaced either before or after. The dimensions of *A* are altered, and the contents are rearranged as appropriate. Additional elements of *A* are taken from the `:initial-element` argument. However, the use of the `:initial-contents` argument causes all old contents to be discarded.

- Suppose A is not displaced before, but is displaced to array C after. None of the original contents of A appears in A afterwards; A now contains (some of) the contents of C, without any rearrangement of C.

- Suppose A is displaced to array B before the call, and is displaced to array C after the call. (Note that B and C may be the same array.) The contents of B do not appear in A afterwards (unless such contents also happen to be in C, as when B and C are the same, for example). If `:displaced-index-offset` is not specified in the call to `adjust-array`, it defaults to zero; the old offset (into B) is not retained.

- Suppose A is displaced to array B before the call, but is not displaced afterwards. In this case A gets a new "data region" and (some of) the contents of B are copied into it as appropriate to maintain the existing old contents. Additional elements of A are taken from the `:initial-element` argument. However, the use of the `:initial-contents` argument causes all old contents to be discarded.

If array X is displaced to array Y, and array Y is displaced to array Z, and array Y is altered by `adjust-array`, array X must now refer to the adjusted contents of Y. This means that an implementation may not collapse the chain to make X refer to Z directly and forget that the chain of reference passes through array Y. (Caching techniques are of course permitted, as long as they preserve the semantics specified here.)

If X is displaced to Y, it is an error to adjust Y in such a way that it no longer has enough elements to satisfy X. This error may be signaled at the time of the adjustment, but this is not required.

Note that omitting the `:displaced-to` argument to `adjust-array` is equivalent to specifying `:displaced-to nil`; in either case, the array is not displaced after the call regardless of whether it was displaced before the call.

Глава 18

Строки

Строка является специализированным вектором (или одномерным массивом), элементы которого — строковые символы.

X3J13 voted in March 1989 to eliminate the type `string-char` and to redefine the type `string` to be the union of one or more specialized vector types, the types of whose elements are subtypes of the type `character`.

Любая функция, определённая в данной главе, имя которой имеет префикс `string`, в качестве аргумента может принимать символ, при условии, что операция не модифицирует этот аргумент. При этом использоваться будет выводимое имя символа. Таким образом операции над последовательностями из строковый символов, не являются специализированными версиями обобщённых функций; обобщённые операции над последовательностями, описанные в главе 14 не принимают символ в качестве последовательности. Такая «неизящность» сделана в Common Lisp'е в целях прагматичности. Для достижения унификации функций, предлагается использовать функцию `string` применительно ко всем аргументам, тип которых не известен заранее.

Note that this remark, predating the design of the Common Lisp Object System, uses the term "generic" in a generic sense and not necessarily in the technical sense used by CLOS (see chapter 2).

Также, существует небольшая расхождение в именах строковых функций. Тогда как суффиксы `equalp` и `eql` были бы более подходящими, вместо них все же для исторической совместимости используются суффиксы `equal` и `=`, для указания, соответственно, регистронезависимого и регистрозависимого сравнения строковых символов.

Любой Lisp'овый объект может быть проверен на принадлежность строковому типу с помощью предиката `stringp`.

Следует отметить, что строки, как и все вектора, могут иметь указатель заполнения (хотя такие строки необязательно простые). Строковые операции в основном взаимодействуют только с активной частью строки (перед указателем заполнения). Смотрите `fill-pointer` и связанные с ним функции.

18.1 Доступ к строковым символам

Следующие функции предоставляют доступ к одиночным символам в строке.

[Функция] **char** *string index*
[Функция] **schar** *simple-string index*

Указываемый индекс *index* должен быть неотрицательным целым меньшим, чем длина строки *string*. Символ из данной позиции строки возвращается, как символьный объект. FIXME

Как и во всех Common Lisp'овых последовательностях, индексация начинается с нуля. Например:

(char "Floob-Boober-Bab-Boober-Bubs" 0) ⇒ #\F
(char "Floob-Boober-Bab-Boober-Bubs" 1) ⇒ #\l

Смотрите `aref` и `elt`. Фактически,

(char s j) ≡ (aref (the string s) j)

`setf` может использоваться с `char` для деструктивной замены символа в строке.

Для `char` строка может быть любой. Для `schar` строка должна быть простой. В некоторых реализациях Common Lisp'а, функция `schar` может быть быстрее, чем `char`.

18.2 Сравнение строк

Правило именования этих функций и их аргументов следует из правил именования общих функций для последовательностей (смотрите главу 14).

[Функция] **string=** *string1 string2* **&key** *:start1 :end1 :start2 :end2*

`string=` сравнивает две строки и возвращает истину, если они одинаковые (все символы одинаковые), и ложь, если это не так. Функция `equal` вызывает `string=`, если применяется к двум строкам.

Именованные аргументы `:start1` и `:start2` указывают на то, с каких позиций начинать сравнение. Именованные аргументы `:end1` и `:end2` указывают на то, на каких позициях заканчивать сравнение. Сравнение заканчивается *перед* указанными позициями. Аргументы «start» по-умолчанию равны нулю (указывают на начало строки), и «end» (если опущены или равны `nil`) по-умолчанию указывают на конец строки. Таким образом, по-умолчанию строки сравниваются целиком. Эти аргументы позволяют удобно сравнивать подстроки. `string=` возвращает ложь, если сравниваемые (под)строки неодинаковой длины. То есть, если

```
(not (= (- end1 start1) (- end2 start2)))
```

истинно, то `string=` ложно.

```
(string= "foo" "foo") истина
(string= "foo" "Foo") ложь
(string= "foo" "bar") ложь
(string= "together" "frog" :start1 1 :end1 3 :start2 2)
  истина
```

X3J13 voted in June 1989 to clarify string coercion (see `string`).

[Функция] **string-equal** *string1 string2* **&key** *:start1 :end1 :start2 :end2*

`string-equal` похож на `string=` за исключением того, что игнорирует регистр символов. Два символа равны если `char-equal` для них истинно. Например:

(string-equal "foo" "Foo") is true

X3J13 voted in June 1989 to clarify string coercion (see `string`).

[Функция] **string<** *string1 string2* **&key** *:start1 :end1 :start2 :end2*
[Функция] **string>** *string1 string2* **&key** *:start1 :end1 :start2 :end2*
[Функция] **string<=** *string1 string2* **&key** *:start1 :end1 :start2 :end2*
[Функция] **string>=** *string1 string2* **&key** *:start1 :end1 :start2 :end2*
[Функция] **string/=** *string1 string2* **&key** *:start1 :end1 :start2 :end2*

Эти функции сравнивают две строки лексикографически. Если не выполняется условие, что *string1* соответственно меньше, больше, меньше либо равно, больше либо равно, не равно чем *string2*, то результат `nil`. Однако, если условие выполняется, то результатом будет первая позиция символа, на которой произошло несовпадение строк. Другими словами, результат это длина префикса строки, удовлетворяющего условию.

Строка a меньше, чем строка b, если в первой позиции, в которой они различаются, символ из a меньше, чем соответствующий символ из b в соответствие с функцией `char<`, или строка a является префиксом строки b (меньшей длины и совпадением всех символов). FIXME

Именованные аргументы `:start1` и `:start2` указывают на то, с каких позиций начинать сравнение. Именованные аргументы `:end1` и `:end2` указывают на то, на каких позициях заканчивать сравнение. Сравнение заканчивается *перед* указанными позициями. Аргументы «start» по-умолчанию равны нулю (указывают на начало строки), и «end» (если опущены или равны `nil`) по-умолчанию указывают на конец строки. Таким образом, по-умолчанию строки сравниваются целиком. Эти аргументы позволяют удобно сравнивать подстроки. Индекс, возвращаемый в случае несовпадения строк, находится в *string1*. FIXME

X3J13 voted in June 1989 to clarify string coercion (see `string`).

[Функция] **string-lessp** *string1 string2* **&key** *:start1 :end1 :start2 :end2*
[Функция] **string-greaterp** *string1 string2* **&key** *:start1 :end1 :start2 :end2*
[Функция] **string-not-greaterp** *string1 string2* **&key** *:start1 :end1 :start2 :end2*
[Функция] **string-not-lessp** *string1 string2* **&key** *:start1 :end1 :start2 :end2*
[Функция] **string-not-equal** *string1 string2* **&key** *:start1 :end1 :start2 :end2*

Эти функции такие же, как и соответственно `string<`, `string>`, `string<=`, `string>=` и `string/=` за исключением того, что различия между регистрами букв игнорируются. Для сравнения символов в таком случае вместо `char<` используется `char-lessp`.

X3J13 voted in June 1989 to clarify string coercion (see `string`).

18.3 Создание и манипулирование строками

Большинство необходимых операций над строками могут быть выполнены с помощью общих функций для последовательностей, описанных в главе 14. Следующие функций производят дополнительные операции специфичные для строк.

[Функция] **make-string** *size* **&key** *:initial-element :element-type*

Функция возвращает простую строку длинной *size*, каждый из символов которой равен аргументу `:initial-element`. Если аргумент `:initial-element` не задан, тогда строка инициализируется в зависимости от реализации.

Аргумент `:element-type` задаёт тип элементов строки. Строка создаётся с наиболее специализированным типов элементов, позволяющем хранить элементы указанного типа. Если `:element-type` опущен, то по-молчанию используется тип `character`.

[Функция] **string-trim** *character-bag string*
[Функция] **string-left-trim** *character-bag string*
[Функция] **string-right-trim** *character-bag string*

`string-trim` возвращает подстроку от *string*, с удалёнными в начале и в конце символами из *character-bag*. Функция `string-left-trim` похожа на предыдущую, но удаляет символы только в начале. `string-right-trim` удаляет символы только в конце. Аргумент *character-bag* может быть любой последовательностью, содержащей строковые символы. Например:

```
(string-trim '(#\Space #\Tab #\Newline) " garbanzo beans
      ") ⇒ "garbanzo beans"
(string-trim " (*)" " ( *three (silly) words* ) ")
  ⇒ "three (silly) words"
(string-left-trim " (*)" " ( *three (silly) words* ) ")
  ⇒ "three (silly) words* ) "
(string-right-trim " (*)" " ( *three (silly) words* ) ")
  ⇒ " ( *three (silly) words"
```

В зависимости от реализации, в случае, если строка не претерпевала модификаций, может возвратиться сама строка или её копия.

X3J13 voted in June 1989 to clarify string coercion (see `string`).

[Функция] **string-upcase** *string* **&key** *:start :end*
[Функция] **string-downcase** *string* **&key** *:start :end*
[Функция] **string-capitalize** *string* **&key** *:start :end*

`string-upcase` возвращает сроку, в которой все символы в нижнем регистре изменены на соответствующие в верхнем. А точнее, все символы строки получаются применением к соответствующим символам входной строки функции `char-upcase`.

`string-downcase` похожа на предыдущую функцию за исключением того, что символы в верхнем регистре заменяются на соответствующие символы в нижнем (используя `char-downcase`).

Именованные аргументы `:start` и `:end` указывают на то, какая часть строки будет обрабатываться. Однако, результат всегда той же длины, что и входная строка.

Аргумент деструктивно не преобразуется. Однако, если символов для перевода регистра в аргументе не оказалось, то на усмотрение реализации результат может быть входной строкой или её копией. Например:

```
(string-upcase "Dr. Livingstone, I presume?")
  ⇒ "DR. LIVINGSTONE, I PRESUME?"
(string-downcase "Dr. Livingstone, I presume?")
  ⇒ "dr. livingstone, i presume?"
(string-upcase "Dr. Livingstone, I presume?" :start 6 :end 10)
  ⇒ "Dr. LiVINGstone, I presume?"
```

`string-capitalize` возвращает копию строки *string*, в которой первые буквы всех слов возводятся в верхний регистр, а остальные буквы в нижний. Слова состоят из алфавитно-цифровых символов, разделённых друг от друга неалфавитно-цифровыми символами. Например:

```
(string-capitalize " hello ") ⇒ " Hello "
(string-capitalize
   "occlUDeD cASEmenTs FOreSTAll iNADVertent DEFenestraTION")
⇒ "Occluded Casements Forestall Inadvertent Defenestration"
(string-capitalize 'kludgy-hash-search) ⇒ "Kludgy-Hash-Search"
(string-capitalize "DON'T!") ⇒ "Don'T!"     ;not "Don't!"
(string-capitalize "pipe 13a, foo16c") ⇒ "Pipe 13a, Foo16c"
```

X3J13 voted in June 1989 to clarify string coercion (see `string`).

[Функция] **nstring-upcase** *string* **&key** *:start :end*
[Функция] **nstring-downcase** *string* **&key** *:start :end*
[Функция] **nstring-capitalize** *string* **&key** *:start :end*

Эти три функции похожи на `string-upcase`, `string-downcase` и `string-capitalize`, но в отличие от последних деструктивно модифицируют аргумент *string* изменяя где надо регистр символов.

Именованные параметры `:start` `:end` задают обрабатываемую подстроку. Результатом является аргумент *string*.

[Функция] **string** *x*

Большинство строковых функций применяют `string` к своим аргументам. Если *x* является строкой, то они и возвращается. Если *x* является символом, то возвращается его выводимое имя.

В любой другой ситуации, сигнализируется ошибка.

Для преобразования последовательности символов в строку используйте `coerce`. (Следует отметить, что `(coerce x 'string)` не сработает, если `x` является Lisp'овым символом. И наоборот, `string` не будет конвертировать список или другую последовательность в строку.)

Для получения строкового отображения для числа или любого другого Lisp'ового объекта используйте `prin1-to-string`, `princ-to-string` или `format`.

X3J13 voted in June 1989 to specify that the following functions perform coercion on their *string* arguments identical to that performed by the function `string`.

string=	string-equal	string-trim
string<	string-lessp	string-left-trim
string>	string-greaterp	string-right-trim
string<=	string-not-greaterp	string-upcase
string>=	string-not-lessp	string-downcase
string/=	string-not-equal	string-capitalize

Note that `nstring-upcase`, `nstring-downcase`, and `nstring-capitalize` are absent from this list; because they modify destructively, the argument must be a string.

As part of the same vote X3J13 specified that `string` may perform additional implementation-dependent coercions but the returned value must be of type `string`. Only when no coercion is defined, whether standard or implementation-dependent, is `string` required to signal an error, in which case the error condition must be of type `type-error`.

Глава 19

Структуры

Common Lisp предоставляет функциональность для создания структур (почти таких же, как в других языках, с именем структуры, полями, и т.д.). Фактически, пользователь может определить новый тип данных.Каждая структура данных этого типа имеет компоненты с заданными именами. При создании структуры автоматически создаются конструктор и конструкции доступа и присваивания значений для полей.

Данная глава разделена на две части. Первая часть описывает основную функциональность структур, которая очень проста и позволяет пользователю воспользоваться проверкой типов, модульностью и удобством определённых им типов данных. Вторая часть начинается с раздела 19.5, описывающего специализированные возможности для сложных приложений. Эти возможности совершенно необязательны к использованию, и вам даже не нужно о них знать для того, чтобы делать хорошие программы.

19.1 Введение в структуры

Функциональность структур воплощена в макросе `defstruct`, который позволяет пользователю создавать и использовать сгруппированные типы данных с именованными элементами. Она похожа на функциональность «структур (structures)» в PL/I или «записей (records)» в Pascal'e.

В качестве примера, предположим, что вы пишете программу на Lisp'e, которая управляет космическими кораблями в двухмерном

пространстве. В вашей программе, вам необходимо представить космический корабль как некоторого вида Lisp'овый объект. Интересующие вашу программу свойства корабля являются: его позиция (представленная как x и y координаты), скорость (представленная как отрезки по осям x и y) и масса.

Таким образом, корабль может быть представлен как запись структуры с пятью компонентами: позиция-x, позиция-y, скорость-x, скорость-y и масса. Эта структура может быть реализована как Lisp'овый объект несколькими способами. Она может быть списком из пяти элементов; позиция-x будет *car* элементом, позиция-y будет *cadr*, и так далее. Подобным образом структура может быть вектором из пяти элементов: позиция-x будет 0-ым элементом, позиция-y будет 1-ым, и так далее. Проблема данных представлений состоит в том, что компоненты занимают совершенно случайные позиции и их сложно запомнить. Кто-нибудь увидев где-то в коде строки `(cadddr ship1)` или `(aref ship1 3)`, обнаружит сложность в определении того, что это производится доступ к компоненту скорости-y структуры `ship1`. Более того, если представление корабля должно быть изменено, то будет очень сложно найти все места в коде для изменения в соответствие с новым представлением (не все появления `cadddr` означают доступ к скорости-y корабля).

Лучше было бы если бы записи структур имели имена. Можно было бы написать что-то вроде `(ship-y-velocity ship1)` вместо `(cadddr ship1)`. Кроме того было бы неплохо иметь более информативную запись для создания структур, чем эта:

```
(list 0 0 0 0 0)
```

Несомненно, хочется, чтобы `ship` был новым типом данных, как и любой другой тип Lisp'овых данных, чтобы, например, осуществить проверку с помощью `typep`. Функциональность `defstruct` предоставляет все, что выше было необходимо.

`defstruct` является макросом, который определяет структуру. Например, для космического корабля, можно определить структуру так:

```
(defstruct ship
  x-position
  y-position
```

```
x-velocity
y-velocity
mass)
```

Запись декларирует, что каждый объект `ship` является объектом с пятью именованными компонентами. Вычисление этой формы делает несколько вещей.

- Она определяет `ship-x-position` как функцию одного аргумента, а именно, корабля, которая возвращает позицию-x корабля. `ship-y-position` и другие компоненты получают такие же определения функций. Эти функции называются *функциями доступа*, так как используются для доступа к компонентам структуры.
- Символ `ship` становится именем типа данных, к которому принадлежат экземпляры объектом кораблей. Например, это имя может использоваться в `typep`. `(typep x 'ship)` истинен, если `x` является кораблём, и ложен, если `x` является любым другим объектом.
- Определяется функция одного аргумента с именем `ship-p`. Она является предикатом, который истинен, если аргумент является кораблём, и ложен в противных случаях.
- Определяется функция с именем `make-ship`, при вызове которой создаётся структура данных из пяти компонентов, готовая к использованию в *функциях доступа*. Так, выполнение

  ```
  (setq ship2 (make-ship))
  ```

  устанавливает в ship2 свеже созданный объект `ship`. Можно указать первоначальные значения для компонентов структуры используя именованные параметры:

  ```
  (setq ship2 (make-ship :mass *default-ship-mass*
                         :x-position 0
                         :y-position 0))
  ```

Форма создаёт новый корабль и инициализирует три его компонента. Эта функция называется *функцией-конструктором*, потому что создаёт новую структуру.

- Синтаксис `#S` может использоваться для чтения экземпляров структур `ship`, и также предоставляется функция вывода для печати структур кораблей. Например, значение ранее упомянутой переменной `ship2` может быть выведено как

  ```
  #S(ship x-position 0 y-position 0 x-velocity nil
        y-velocity nil mass 170000.0)
  ```

- Определяется функция одного аргумента с именем `copy-ship`, которая получая объект `ship`, будет создавать новый объект `ship`, который является копией исходного. Эта функция называется *функцией копирования*.

- Можно использовать `setf` для изменения компонентов объекта `ship`:

  ```
  (setf (ship-x-position ship2) 100)
  ```

  Данная запись изменяет позицию-x переменной *ship2* в `100`. Она работает, потому что `defstruct` ведёт себя так, будто создаёт соответствующие `defsetf` формы для *функций доступа*.

Этот простой пример отображает мощь и удобство `defstruct` для представления записей структур. `defstruct` имеет много других возможностей для специализированных целей.

19.2 Как использовать defstruct

Все структуры определяются с помощью конструкции `defstruct`. Вызов `defstruct` определяет новый тип данных, экземпляры которого содержат именованные слоты.

[Макрос] **defstruct** name-and-options [doc-string] {slot-description}+

Этот макрос определяет структурный тип данных. Обычно вызов `defstruct` выглядит как следующий пример.

(defstruct (*name option-1 option-2 ... option-m*)
doc-string
slot-description-1
slot-description-2
...
slot-description-n)

Имя структуры *name* должно быть символом. Он становиться именем нового типа данных, который включает в себя все экземпляры данной структуры. Соответственно функция `typep` будет принимать и использовать это имя. Имя структуры *name* возвращается как значение формы *defstruct*.

X3J13 voted in June 1988 to allow a `defstruct` definition to have no *slot-description* at all; in other words, the occurrence of {slot-description}+ in the preceding header line would be replaced by {slot-description}* .

Such structure definitions are particularly useful if the `:include` option is used, perhaps with other options; for example, one can have two structures that are exactly alike except that they print differently (having different `:print-function` options).

Implementors are encouraged to permit this simple extension as soon as convenient. Users, however, may wish to maximize portability of their code by avoiding the use of this extension unless and until it is adopted as part of the ANSI standard.

Обычно опции не нужны. Если они не заданы, то после слова `defstruct` вместо `(`*`name`*`)` можно записать просто *name* . Синтаксис опций и их смысл раскрываются в разделе 19.5.

Если присутствует необязательная строка документации *doc-string*, тогда она присоединяется к символу *name*, как документация для типа `structure`. Смотрите `documentation`.

Каждое описание слота *slot-description-j* выглядит так:

(*slot-name default-init*

slot-option-name-1 slot-option-value-1
slot-option-name-2 slot-option-value-2
...
slot-option-name-k_j slot-option-value-k_j)

Каждое имя слота *slot-name* должно быть символом. Для каждого слота определяется функция доступа. Если опции и первоначальные значения не указаны, тогда в качестве описания слота вместо `(slot-name)` можно записать просто *slot-name*.

Форма *default-init* вычисляется, только если соответствующий аргумент не указан при вызове функций-конструкторов. *default-init* вычисляется *каждый раз*, когда необходимо получить первоначальное значение для слота.

Если *default-init* не указано, тогда первоначальное значение слота не определено и зависит от реализации. Доступные опции для слотов разобраны в разделе 19.4.

X3J13 voted in January 1989 to specify that it is an error for two slots to have the same name; more precisely, no two slots may have names for whose print names `string=` would be true. Under this interpretation

(defstruct lotsa-slots slot slot)

obviously is incorrect but the following one is also in error, even assuming that the symbols `coin:slot` and `blot:slot` really are distinct (non-`eql`) symbols:

(defstruct no-dice coin:slot blot:slot)

To illustrate another case, the first `defstruct` form below is correct, but the second one is in error.

(defstruct one-slot slot)
(defstruct (two-slots (:include one-slot)) slot)

Обоснование: Print names are the criterion for slot-names being the same, rather than the symbols themselves, because `defstruct` constructs names of accessor functions from the print names and interns the resulting new names in the current package.

X3J13 recommended that expanding a `defstruct` form violating this restriction should signal an error and noted, with an eye to the Common Lisp Object System , that the restriction applies only to the operation of the `defstruct` macro as such and not to the `structure-class` or structures defined with `defclass`.

X3J13 voted in March 1989 to clarify that, while defining forms normally appear at top level, it is meaningful to place them in non-top-level contexts; `defstruct` must treat slot *default-init* forms and any initialization forms

within the specification of a by-position constructor function as occurring within the enclosing lexical environment, not within the global environment.

`defstruct` не только определяет функции доступа к слотам, но также выполняет интеграцию с `setf` для этих функций, определяет предикат с именем *name*`-p`, определяет функцию-конструктор с именем `make-`*name* и определяет функцию копирования с именем `copy-`*name*. Все имена автоматически создаваемых функций интернируются в текущий пакет на время выполнения `defstruct` (смотрите `*package*`). Кроме того, все эти функции могут быть задекларированы как `inline` на усмотрение реализации для повышения производительности. Если вы не хотите, чтобы функции были задекларированы как `inline`, укажите декларацию `notinline` после формы `defstruct` для перезаписи декларации `inline`.

X3J13 voted in January 1989 to specify that the results of redefining a `defstruct` structure (that is, evaluating more than one `defstruct` structure for the same name) are undefined.

The problem is that if instances have been created under the old definition and then remain accessible after the new definition has been evaluated, the accessors and other functions for the new definition may be incompatible with the old instances. Conversely, functions associated with the old definition may have been declared `inline` and compiled into code that remains accessible after the new definition has been evaluated; such code may be incompatible with the new instances.

In practice this restriction affects the development and debugging process rather than production runs of fully developed code. The `defstruct` feature is intended to provide "the most efficient" structure class. CLOS classes defined by `defclass` allow much more flexible structures to be defined and redefined.

Programming environments are allowed and encouraged to permit `defstruct` redefinition, perhaps with warning messages about possible interactions with other parts of the programming environment or memory state. It is beyond the scope of the Common Lisp language standard to define those interactions except to note that they are not portable.

19.3 Использование автоматически определяемого конструктора

После того, как вы определили новую структуру с помощью `defstruct`, вы можете создавать экземпляры данной структуры с помощью функции-конструктора. По-умолчанию `defstruct` автоматически определяет эту функцию. Для структуры с именем `foo`, функция-конструктор обычно называется `make-foo`. Вы можете указать другое имя, передав его в качестве аргумента для опции `:constructor`, или, если вы вообще не хотите обычную функцию-конструктор, указав в качества аргумента `nil`. В последнем случае должны быть запрошены один или более конструкторов с «позиционными» аргументами FIXME, смотрите раздел 19.6.

Общая форма вызова функции-конструктора выглядит так:

(*name-of-constructor-function*
    slot-keyword-1 form-1
    slot-keyword-2 form-2
    ...)

Все аргументы являются именованными. Каждый *slot-keyword* должен быть ключевым символом, имя которого совпадает со именем слота структуры (`defstruct` определяет возможные ключевые символы интернируя каждое имя слота в пакет ключевых символов (keyword). Все ключевые символы *keywords* и формы *forms* вычисляются. В целом, это выглядит как если функция-конструктор принимает все аргументы как `&key` параметры. Например, структура `ship` упомянутая ранее в разделе 19.1 имеет функцию-конструктор, которая принимает аргументы в соотвествие со следующим определением:

```
(defun make-ship (&key x-position y-position
                       x-velocity y-velocity mass)
  ...)
```

Если *slot-keyword-j* задаёт имя слота, тогда элемент созданной структуры будет инициализирован значением *form-j*. Если пара *slot-keyword-j* и *form-j* для слота не указана, тогда слот будет

инициализирован результатом вычисления указанной в вызове `defstruct` для этого слота формой *default-init*. (Другими словами, инициализация указанная в `defstruct` замещается инициализацией указанной в вызове функции-конструктора.) Если используется форма инициализации *default-init*, она вычисляется в время создания экземпляра структуры, но в лексическом окружении формы `defstruct`, в которой она используется. Если эта форма инициализации не указана, первоначальное значение слота не определено. Если вам необходимо инициализировать слоты некоторыми значениями, вы должны всегда указывать первоначальное значение или в `defstruct`, или в вызове функции-конструктора.

Каждая форма инициализации, указанная в компоненте формы `defstruct`, когда используется функцией-конструктором, перевычисляется при каждом вызове функции. Это, как если бы формы инициализации использовались как формы *init* для именованных параметров функции-конструктора. Например, если форма `(gensym)` используется как форма инициализации или в вызове функции-конструктора, или форме инициализации слота в форме `defstruct`, тогда каждый вызов функции-конструктора вызывал бы `gensym` для создания нового символа.

X3J13 voted in October 1988 to clarify that the default value in a defstruct slot is not evaluated unless it is needed in the creation of a particular structure instance. If it is never needed, there can be no type-mismatch error, even if the type of the slot is specified, and no warning should be issued.

For example, in the following sequence only the last form is in error.

```
(defstruct person (name .007 :type string))

(make-person :name "James")

(make-person)     ;Error to give name the value .007
```

19.4 Опции слотов для defstruct

Каждая форма *slot-description* в форме `defstruct` может указывать одно или более опций слота. *slot-option* является парой ключевого символа и значения (которое не является формой для вычисления, а является просто значением). Например:

```
(defstruct ship
  (x-position 0.0 :type short-float)
  (y-position 0.0 :type short-float)
  (x-velocity 0.0 :type short-float)
  (y-velocity 0.0 :type short-float)
  (mass *default-ship-mass* :type short-float :read-only t))
```

Этот пример содержит определение, что каждый слот будет всегда содержать короткое с плавающей точкой число, и что последний слот не может быть изменён после создания корабля. Доступные опции для слотов *slot-options*:

`:type` Опция `:type` ***type*** указывает, что содержимое слота будет всегда принадлежать указанному типу данных. Она похожа на аналогичную декларацию для переменной или функции. И конечно же, она также декларирует возвращаемый *функцией доступа* тип. Реализация может проверять или не проверять тип нового объекта при инициализации или присваивании слота. Следует отметить, что форма аргумента *type* не вычисляется, и следовательно должна быть корректным спецификатором типа.

`:read-only` Опция `:read-only` ***x***, при x не `nil`, указывает , что этот слот не может быть изменён. Он будет всегда содержать значение, указанное во время создания экземпляра структуры. `setf` не принимает *функцию доступа* к данному слоту. Если x `nil`, эта опция ничего не меняет. Следует отметить, что форма аргумента x не вычисляется.

Следует отметить, что невозможно определить опцию для слота без указания значение по-умолчанию.

19.5 Опции defstruct

Предыдущее описание `defstruct` достаточно для среднестатистического использования. Оставшуюся часть этой главы занимает описание более сложных возможностей функционала `defstruct`.

Данный раздел объясняет каждую опцию, которая может быть использована в `defstruct`. Опция для `defstruct` может быть или ключевым символом, или списком из ключевого символа и аргумента для него. (Следует отметить, что синтаксис для опций `defstruct` отличается от синтаксиса пар, используемых для опций слота. Никакая часть этих опций не вычисляется.)

`:conc-name` Данная опция предоставляет префикс для имён функций доступа. По соглашению, имена всех функций доступа к слотам структура начинаются с префикса — имени структуры с последующим дефисом. Это поведение по-умолчанию.

Аргумент `:conc-name` указывает альтернативный префикс. (Если в качестве разделителя используется дефис, он указывается как часть префикса.) Если в качестве аргумента указано `nil`, тогда префикс не устанавливается вообще. Тогда имена функций доступа совпадают с именами слотов, и это повод давать слотам информативные имена.

Следует отметить, что не зависимо от того, что указана в `:conc-name`, в функции-конструкторе используются ключевые символы, совпадающие с именами слотов без присоединяемого префикса. С другой стороны префикс используется в именах функций доступа. Например:

```
(defstruct door knob-color width material)
(setq my-door
      (make-door :knob-color 'red :width 5.0))
(door-width my-door) ⇒ 5.0
(setf (door-width my-door) 43.7)
(door-width my-door) ⇒ 43.7
(door-knob-color my-door) ⇒ red
```

`:constructor` Данная опция принимает один аргумент, символ, который указывает имя функции-конструктора. Если аргумент не предоставлен, или если опция не указана, имя конструктора создаётся соединением строки `"MAKE-"` и имени структуры, и помещается в текущий пакет в время выполнения формы `defstruct` (смотрите `*package*`). Если аргумент указан и равен `nil`, то функция конструктор не создаётся.

Эта опция имеет более сложный синтаксис, описываемый в разделе 19.6.

`:copier` Данная опция принимает один аргумент, символ, который указывает имя функции копирования. Если аргумент не предоставлен, или если опция не указана, имя функции копирования создаётся соединением строки `"COPY-"` и имени структуры, и помещается в текущий пакет в время выполнения формы `defstruct` (смотрите `*package*`). Если аргумент указан и равен `nil`, то функция копирования не создаётся.

Автоматически создаваемая функция копирования просто создаёт новую структуру и переносит все компоненты из структуры аргумента в свеже создаваемую структуру. Копирование самих компонентов структуры не производится. Соответствующие элементы старой и новой структуры равны `eql` между собой.

`:predicate` Эта опция принимает один аргумент, который задаёт имя предиката типа. Если аргумент не указан, или если не указана опция, то имя предиката создаётся соединением имени структуры и строки `"-P"`, помещая имя в текущий пакет на момент вычисления формы `defstruct` (смотрите `*package*`). Если указанный аргумент равен `nil`, предикат не создаётся. Предикат может быть определён, только если структура имеет «имя». Если указана опция `:type` и не указана `:named`, тогда опция `:predicate` не должна использоваться или должна иметь значение `nil`.

`:include` Эта опция используется для создания нового определения структуры как расширения для старого определения структуры. В качестве примера, предположим у вас есть структура, называемая `person`, которая выглядит так:

```
(defstruct person name age sex)
```

Теперь, предположим, вы хотите создать новую структуру для представления астронавта. Так как астронавт также человек, вы хотите, чтобы он также имел свойства имя, возраст и пол, и вы хотите чтобы Lisp'овые функции работали со структурами `astronaut` также как и с структурами `person`. Вы можете сделать это определив структуру `astronaut` с опцией `:include`, так:

```
(defstruct (astronaut (:include person)
                      (:conc-name astro-))
  helmet-size
  (favorite-beverage 'tang))
```

Опция `:include` заставляет структуру, будучи определённой, иметь те же слоты, что и включаемая структура из опции. Это осуществляется с помощью того, что функции доступа включаемой структуры будут также работать с определяемой структурой. Таким образом, в этом примере, `astronaut` будет иметь пять слотов: три определены в `person` и два в самом `astronaut`. Функции доступа, определённые с помощью структуры `person`, могут применяться к экземплярам структуры `astronaut`, и будут корректно работать. Более того, `astronaut` будет иметь свои функции доступа для компонентов унаследованных от структуры `person`. Следующий пример иллюстрирует то, как вы можете использовать структуры `astronaut`:

```
(setq x (make-astronaut :name 'buzz
                        :age 45
                        :sex t
                        :helmet-size 17.5))

(person-name x) ⇒ buzz
(astro-name x) ⇒ buzz

(astro-favorite-beverage x) ⇒ tang
```

Различие между функциями доступа `person-name` и `astro-name` в том, что `person-name` может быть корректно применена к любому экземпляру `person`, включая `astronaut`, тогда как `astro-name` может работать только с `astronaut`. (Реализация может проверять или не проверять корректное использоваться таких функций доступа.)

В одной форме `defstruct` может использовать не более одной опции `:include`. Аргумент для опции `:include` является обязательным и должен быть именем определённой ранее структуры. Если структура, будучи определённой, не имела опции `:type`, тогда наследуемая структура также не должна была содержать опцию `:type`. Если структура, будучи определённой, имела опцию `:type`, тогда наследуемая структура должна была иметь опцию `:type` с тем же типом.

Если опция `:type` не указана, тогда имя структуры становиться именем типа данных. Более того, тип будет является подтипом типа структуры, от которой произошло наследование. В вышеприведённом примере, `astronaut` является подтипом `person`. Так,

```
(typep (make-astronaut) 'person)
```

но и указывает, что все операции над `person` будут также работать для `astronaut`.

Далее рассказывается чуть более сложные возможности опции `:include`. Иногда, когда одна структура включает другую, необходимо, чтобы значения по-умолчанию или опции слотов из родительской структуры при наследовании стали слегка другими. Новая структура может задать значения по-умолчанию или опции для наследуемых слотов отличными от родительских, с помощью конструкции:

(:include *name slot-description-1 slot-description-2 ...*)

Каждая форма *slot-description-j* должна иметь имя *slot-name* или *slot-keyword*, такое же как в родительской структуре. Если *slot-description-j* не имеет формы инициализации *default-init*, тогда в новой структуре слот также не будет иметь первоначального значения. В противном случае его первоначальное значение будет заменено формой *default-init* из *slot-description-j*. Доступный для записи слот может быть переделан в слот только для чтения. Если слот только для чтения в родительской структуре, тогда он также должен быть только для чтения в дочерней. Если для слота указан тип, он должен быть таким же или подтипом в дочерней структуре. Если это строгий подтип, то реализация может проверять или не проверять ошибки несовпадения типов при присваивании значений слотам.

Например, если мы хотели бы определить `astronaut` так, чтобы по-умолчанию возрастом астронавта было 45 лет, то мы могли бы сказать:

```
(defstruct (astronaut (:include person (age 45)))
  helmet-size
  (favorite-beverage 'tang))
```

X3J13 voted in June 1988 to require any structure type created by `defstruct` (or `defclass`) to be disjoint from any of the types `cons`, `symbol`, `array`, `number`, `character`, `hash-table`, `readtable`, `package`, `pathname`, `stream`, and `random-state`. A consequence of this requirement is that it is an error to specify any of these types, or any of their subtypes, to the `defstruct :include` option. (The first edition said nothing explicitly about this. Inasmuch as using such a type with the `:include` option was not defined to work, one might argue that such use was an error in Common Lisp as defined by the first edition.)

`:print-function` Эта опция может использоваться, только если не опция `:type` не была указана. Аргумент для опции `:print-function` должен быть функцией трёх аргументов, в форме принимаемой специальной формой `function`. Эта функция используется для вывода рассматриваемой структуры. Когда

структура выводится на консоль, данная функция вызывается с тремя аргументами: структура для печати, поток, в который производить вывод, и целое число, отображающее текущую глубину (можно сравнить с `*print-level*`). Функция вывода должна следить за значениями таких переменных настройки вывода, как `*print-escape*` и `*print-pretty*`.

Если опции `:print-function` и `:type` не указаны, тогда функция вывода по-умолчанию выводит все слоты используя синтаксис `#S` (смотрите раздел 22.1.4).

X3J13 voted in January 1989 to specify that user-defined printing functions for the `defstruct :print-function` option may print objects to the supplied stream using `write`, `print1`, `princ`, `format`, or `print-object` and expect circularities to be detected and printed using `#n#` syntax (when `*print-circle*` is non-`nil`, of course). See `*print-circle*`.

X3J13 voted in January 1989 to clarify that if the `:print-function` option is not specified but the `:include` option *is* specified, then the print function is inherited from the included structure type. Thus, for example, an `astronaut` will be printed by the same printing function that is used for `person`.

X3J13 in the same vote extended the `print-function` option as follows: If the `print-function` option is specified but with no argument, then the standard default printing function (that uses `#S` syntax) will be used. This provides a means of overriding the inheritance rule. For example, if `person` and `astronaut` had been defined as

```
(defstruct (person
             (:print-function     ;Special print function
             (lambda (p s k)
               (format s "<~A, age ~D>"
                       (person-name p)
                       (person-age p)))))
  name age sex)
```

```
(defstruct (astronaut
             (:include person)
             (:conc-name astro-)
             (:print-function))     ;Use default print function
  helmet-size
  (favorite-beverage 'tang))
```

then an ordinary person would be printed as "`<Joe Schmoe, age 27>`" but an astronaut would be printed as, for example,

```
#S(ASTRONAUT NAME BUZZ AGE 45 SEX T
  HELMET-SIZE 17.5 FAVORITE-BEVERAGE TANG)
```

using the default `#S` syntax (yuk).

These changes make the behavior of `defstruct` with respect to the `:include` option a bit more like the behavior of classes in CLOS.

`:type` Опция `:type` явно задаёт представление используемое для структуры. Она принимает один аргумент, который должен быть одним из перечисленных ниже.

Задание этой опции заставляет использовать указанное представление и заставляет компоненты быть размещёнными в порядке, предусмотренном в `defstruct` форме, в соответствующие последовательные элементы указанного представления. Опция также отключает возможность имени структуры стать именем типа. (смотрите раздел 19.7).

Обычно эта опция не используется, и тогда структура представляется так, как предусмотрено в реализации.

`vector` Структура представляется как вектор `(vector t)`. размещая компоненты как элементы вектора. Первый компонент находится в первом элементе вектора, если структура содержит опцию `:named`, и в нулевом, если структура содержит `:unnamed`.

`(vector element-type)` Структура представляется как (возможно специализированный) вектор, размещая компоненты как элементы вектора. Каждый компонент должен принадлежать типу, который мог быть указан для вектора. Первый компонент находится в первом элементе вектора, если структура содержит опцию `:named`, и в нулевом, если структура содержит `:unnamed`. Структура может быть `:named`, только если тип `symbol` является подтипом указанного в `element-type`.

`list` Структура представляется как список. Первый компонент является *cadr* элементом, если структура содержит `:named`, и является *car* элементом, если структура содержит `:unnamed`.

`:named` Опция `:named` указывает, что структура будет иметь «имя». Эта опция не принимает аргументов. Если опция `:type` не указана, тогда структура имеет имя. Таким образом эта опция полезно только при использовании вместе опцией `:type`. Смотрите раздел 19.7 для подробного описания этой опции.

`:initial-offset` Эта опция позволяет вам указать `defstruct` пропустить указанное количество слотов, перед тем как начинать размещать указанные в теле слоты. Эта опция требует аргумент, а именно, неотрицательное целое, обозначающее количество пропускаемых слотов. Опция `:initial-offset` может использоваться, только если также указана опция `:type`. Смотрите раздел **??** для подробного описания этой опции.

19.6 Функции-конструкторы с позиционными аргументами

Если опция `:constructor` указана так (`:constructor name arglist`), тогда вместо создания конструктора с именованными параметрами, `defstruct` определит конструктор с позиционными аргументами.

Форма *arglist* используется для описания того, какие аргументы будет принимать конструктор. В простейшем случае что-то вроде `(:constructor make-foo (a b c))` определяет функцию `make-foo` с тремя аргументами, которые используются для инициализации слотов `a`, `b` и `c`.

Кроме того в список аргументов могут использоваться `&optional`, `&rest` и `&aux`. Они работают так, как и ожидается, но есть несколько тонкостей достойных объяснения. Рассмотрим пример:

```
(:constructor create-foo
      (a &optional b (c 'sea) &rest d &aux e (f 'eff)))
```

Эта конструкция определяет конструктор `create-foo` для использования с одним или более аргументами. Первый аргумент используется для инициализации слота `a`. Если второй параметр не указан, используется значение (если указано) по-умолчанию из тела `defstruct`. Третий аргумент используется для инициализации слота `c`. Если третий параметр не указан, тогда используется символ `sea`. Все параметры после третьего собираются в список и используются для инициализации слота `d`. Если указано три и более параметров, тогда в слот `d` помещается значение `nil`. Слот `e` *не инициализируется*. Его первоначальное значение не определено. Наконец, слот `f` инициализируется символом `eff`.

Действия со слотами `b` и `e` выбраны не случайно, а для того чтобы показать все возможные случаи использования аргументов. Следует отметить, что `&aux` (вспомогательные) «переменные» могут использоваться для перекрытия форм инициализации из тела `defstruct`.

Следуя этому определению можно записать

```
(create-foo 1 2)
```

вместо

```
(make-foo :a 1 :b 2)
```

и, конечно, `create-foo` предоставляет инициализацию отличную от `make-foo`.

Использовать `:constructor` можно более одного раза. Таким образом вы можете определить несколько различных функций-конструкторов с различными аргументами.

X3J13 voted in January 1989 to allow `&key` and `&allow-other-keys` in the parameter list of a "positional" constructor. The initialization of slots corresponding to keyword parameters is performed in the same manner as for `&optional` parameters. A variant of the example shown above illustrates this:

```
(:constructor create-foo
      (a &optional b (c 'sea)
       &key p (q 'cue) ((:why y)) ((:you u) 'ewe)
       &aux e (f 'eff)))
```

The treatment of slots `a`, `b`, `c`, `e`, and `f` is the same as in the original example. In addition, if there is a `:p` keyword argument, it is used to initialize the `p` slot; if there isn't any `:p` keyword argument, then the default value given in the body of the `defstruct` (if given) is used instead. Similarly, if there is a `:q` keyword argument, it is used to initialize the `q` slot; if there isn't any `:q` keyword argument, then the symbol `cue` is used instead.

In order thoroughly to flog this presumably already dead horse, we further observe that if there is a `:why` keyword argument, it is used to initialize the `y` slot; otherwise the default value for slot `y` is used instead. Similarly, if there is a `:you` keyword argument, it is used to initialize the `u` slot; otherwise the symbol `ewe` is used instead.

If memory serves me correctly, `defstruct` was included in the original design for Common Lisp some time before keyword arguments were approved. The failure of positional constructors to accept keyword arguments may well have been an oversight on my part; there is no logical reason to exclude them. I am grateful to X3J13 for rectifying this.

A remaining difficulty is that the possibility of keyword arguments renders the term "positional constructor" a misnomer. Worse yet, it ruins the term "BOA constructor." I suggest that they continue to be called BOA constructors, as I refuse to abandon a good pun. (I regret appearing to have more compassion for puns than for horses.)

As part of the same vote X3J13 also changed `defstruct` to allow BOA constructors to have parameters (including supplied-p parameters) that do not correspond to any slot. Such parameters may be used in subsequent initialization forms in the parameter list. Consider this example:

```
(defstruct (ice-cream-factory
             (:constructor fabricate-factory
               (&key (capacity 5)
                     location
                     (local-flavors
                       (case location
                         ((hawaii) '(pineapple macadamia guava))
                         ((massachusetts) '(lobster baked-bean))
                         ((california) '(ginger lotus avocado
                                         bean-sprout garlic))
                         ((texas) '(jalapeno barbecue))))
                     (flavors (subseq (append local-flavors
                                              '(vanilla
                                                chocolate
                                                strawberry
                                                pistachio
                                                maple-walnut
                                                peppermint))
                                      0 capacity)))))
  (capacity 3)
  (flavors '(vanilla chocolate strawberry mango)))
```

The structure type `ice-cream-factory` has two constructors. The standard constructor, `make-ice-cream-factory`, takes two keyword arguments named `:capacity` and `:flavors`. For this constructor, the default for the `capacity` slot is `3` and the default list of `flavors` is America's favorite threesome and a dark horse (not a dead one). The BOA constructor `fabricate-factory` accepts four different keyword arguments. The `:capacity` argument defaults to `5`, and the `:flavors` argument defaults in a complicated manner based on the other three. The `:local-flavors` argument may be specified directly, or may be allowed to default based on the `:location` of the factory. Here are examples of various factories:

```
(setq houston (fabricate-factory :capacity 4 :location 'texas))
(setq cambridge (fabricate-factory :location 'massachusetts))
(setq seattle (fabricate-factory :local-flavors '(salmon)))
(setq wheaton (fabricate-factory :capacity 4 :location 'illinois))
(setq pittsburgh (fabricate-factory :capacity 4))
(setq cleveland (make-factory :capacity 4))

(ice-cream-factory-flavors houston)
 ⇒ (jalapeno barbecue vanilla chocolate)
```

(ice-cream-factory-flavors cambridge)
⇒ (lobster baked-bean vanilla chocolate strawberry)

(ice-cream-factory-flavors seattle)
⇒ (salmon vanilla chocolate strawberry pistachio)

(ice-cream-factory-flavors wheaton)
⇒ (vanilla chocolate strawberry pistachio)

(ice-cream-factory-flavors pittsburgh)
⇒ (vanilla chocolate strawberry pistachio)

(ice-cream-factory-flavors cleveland)
⇒ (vanilla chocolate strawberry mango)

19.7 Структуры с явно заданным типом представления

Иногда необходимо явно контролировать представление структуры. Опция `:type` позволяет выбрать представление между списком или некоторым видом вектора и указать соотвествие для размещения слотов в выбранном представлении. Структура также может быть «безымянной» или «именованной». Это означает может ли имя структуры быть сохранено в ней самой (и соответственно, прочитано из неё).

19.7.1 Безымянные структуры

Иногда конкретное представление данных навязывается внешними требованиями и, кроме того, формат данных прекрасно ложится в структурный стиль хранения. Например, рассмотрим выражение созданное из чисел, символов и таких операций как `+` и `*`. Операция может быть представлена, как в Lisp'е, списком из оператора и двух

операндов. Этот факт может быть выражен кратко в терминах `defstruct`: e

```
(defstruct (binop (:type list))
  (operator '? :type symbol)
  operand-1
  operand-2)
```

Результатом выполнения `make-binop` является 3-ёх элементный список:

```
(make-binop :operator '+ :operand-1 'x :operand-2 5)
  ⇒ (+ x 5)

(make-binop :operand-2 4 :operator '*)
  ⇒ (* nil 4)
```

Выглядит как функция `list` за исключением того, что принимает именованные параметры и выполняет инициализацию слотов соответствующую концептуальному типу данных `binop`. Таким же образом, селекторы `binop-operator`, `binop-operand-1` и `binop-operand-2` эквивалентны соответственно `car`, `cadr` и `caddr`. (Они, конечно, не полностью эквивалентны, так как реализация может осуществлять проверки типов элементов, длины массивов при использовании селекторов слотов структур.)

Мы говорим о `binop` как о «концептуальном» типе данных, потому что `binop` не принадлежит Common Lisp'овой системе типов. Предикат `typep` не может использовать `binop` как спецификатор типа, и `type-of` будет возвращать `list` для заданной `binop` структуры. Несомненно, различий между структурой данных, созданной с помощью `make-binop`, и простым списком нет.

Невозможно даже получить имя структуры для объекта, созданного с помощью `make-binop`. Однако имя может быть сохранено и получено, если структура содержит «имя».

19.7.2 Именованные структуры

Структура с «именем» имеет свойство, которое заключается в том, что для любого экземпляра структуры можно получить имя этой структуры. Для структур определённых без указания опции `:type`, имя структуры фактически становится частью Common Lisp'овой системы типов. Функция `type-of` при применении к экземпляру такой структуры будет возвращать имя структуры. Предикат `typep` будет рассматривать имя структуры, как корректный спецификатор типа.

Для структур определённых с опцией `:type`, `type-of` будет возвращать спецификатор типа один из `list` или `(vector t)`, в зависимости от указанного аргумента опции `:type`. Имя структуры не становится корректным спецификатором типа. Однако если также указана опция `:named`, тогда первый компонент структуры всегда содержит её имя. Это позволяет получить это имя, имея только экземпляр структуры. Это также позволяет автоматически определить предикат для концептуального типа. Предикат *`name`*`-p` для структуры принимает в качестве первого аргумента объект и истинен, если объект является экземпляром структуры, иначе ложен.

Рассмотрим вышеупомянутый пример `binop` и модифицируем его, добавив опцию `:named`:

```
(defstruct (binop (:type list) :named)
  (operator '? :type symbol)
  operand-1
  operand-2)
```

Как и раньше, конструкция определить функцию-конструктор `make-binop` и три функции-селектора `binop-operator`, `binop-operand-1` и `binop-operand-2`. Она также определит предикат `binop-p`.

Результатом `make-binop` теперь является список с 4-мя элементами:

```
(make-binop :operator '+ :operand-1 'x :operand-2 5)
  ⇒ (binop + x 5)

(make-binop :operand-2 4 :operator '*)
  ⇒ (binop * nil 4)
```

Структура имеет такую же разметку как и раньше за исключением имени структуры в первом элементе. Функции-селекторы `binop-operator`, `binop-operand-1`и `binop-operand-2` эквивалентны соответственно `cadr`, `caddr` и `cadddr`. Предикат `binop-p` примерно соответствует следующему определению.

```
(defun binop-p (x)
  (and (consp x) (eq (car x) 'binop)))
```

Имя `binop` не является корректным спецификатором типа, и не может использоваться в `typep`. Но с помощью предиката структура теперь отличима от других структур.

19.7.3 Другие аспекты явно определённых типов для представления структур

Опция `:initial-offset` позволяет указывать начало размещения слотов в представлении структуры. Например, форма

```
(defstruct (binop (:type list) (:initial-offset 2))
  (operator '? :type symbol)
  operand-1
  operand-2)
```

создаст конструктор `make-binop` со следующим поведением:

```
(make-binop :operator '+ :operand-1 'x :operand-2 5)
  ⇒ (nil nil + x 5)

(make-binop :operand-2 4 :operator '*)
  ⇒ (nil nil * nil 4)
```

Селекторы `binop-operator`, `binop-operand-1` и `binop-operand-2` будут эквивалентны соответственно `caddr`, `cadddr`, и `car` от `cddddr`. Таким же образом, форма

```
(defstruct (binop (:type list) :named (:initial-offset 2))
  (operator '? :type symbol)
  operand-1
  operand-2)
```

создаст конструктор `make-binop` со следующим поведением:

```
(make-binop :operator '+ :operand-1 'x :operand-2 5)
  ⇒ (nil nil binop + x 5)

(make-binop :operand-2 4 :operator '*)
  ⇒ (nil nil binop * nil 4)
```

Если вместе с опцией `:type` используется `:include`, тогда в представлении выделяется столько места, сколько необходимо для родительской структуры, затем пропускается столько места, сколько указано в опции `:initial-offset`, и затем начинается расположение элементов определяемой структуры. Например:

```
(defstruct (binop (:type list) :named (:initial-offset 2))
  (operator '? :type symbol)
  operand-1
  operand-2)

(defstruct (annotated-binop (:type list)
                            (:initial-offset 3)
                            (:include binop))
  commutative associative identity)

(make-annotated-binop :operator '*
                      :operand-1 'x
                      :operand-2 5
                      :commutative t
```

```
                :associative t
                :identity 1)
⇒ (nil nil binop * x 5 nil nil nil t t 1)
```

Первые два `nil` элемента пропущены по причине опции `:initial-offset` со значением `2` в определении `binop`. Следующие четыре элемента содержат имя и три слота структуры `binop`. Следующие три `nil` элементы пропущено по причине опции `:initial-offset` за значением `3` в определении структуры `annotated-binop`. Последние три элемента содержат три слота, определённых в структуре `annotated-binop`.

Глава 20

Вычислитель

Механизм, который выполняет Lisp'овые программы, называется вычислитель. А точнее, вычислитель принимает форму и выполняет расчёты указанные формой. Этот механизм доступен пользователю через функцию `eval`.

Вычислитель, как правило, реализован как интерпретатор, который рекурсивно проходит по заданной форме, выполняя каждый шаг вычисления. Однако такая реализация не обязательна. Возможно альтернативное поведение, когда вычислитель сначала полностью компилирует форму в машинновыполняемый код, и затем его вызывает. Этот метод практически исключает несовместимости между интерпретируемым и компилируемым кодом, но и делает механизм `evalhook` относительно бесполезным. Допустимы также разные смешанные стратегии. Все эти методы должны возвращать одинаковые результаты для правильного кода, и могут возвращать разные ошибки для неправильного кода. Например, поведение может отличаться в том, когда раскрывается макровызов. Поэтому определение макроса не должно зависеть от времени, когда он раскрывается. Для каждой реализации разработчики должны документировать стратегию вычисления.

20.1 Вычисление форм

Функция `eval` является главным пользовательским интерфейсом к вычислителю. Для пользовательских отладочных функций в

интерпретаторе предусмотрены ловушки. Функции `evalhook` и `applyhook` предоставляют альтернативные интерфейсы к механизму вычислителя для использования этих отладочных возможностей.

[Функция] **eval** *form*

Форма *form* вычисляется в текущем динамическом окружении и нулевом лексическом. Результатом функции является вычисленное значение для переданной формы.

Следует отметить, что когда вы записываете вызов к `eval`, то для переданной формы происходят *два* уровня вычислений. Сначала происходит вычисление формы аргумента, как и любого аргумента для любой функции. Данное вычисление в свою очередь неявно вызывает `eval`. Затем происходит вычисление значения аргумента переданного в функцию `eval`. Например:

(eval (list 'cdr (car '((quote (a . b)) c)))) ⇒ b

Форма аргумента `(list 'cdr (car '((quote (a . b)) c)))` вычисляется в `(cdr (quote (a . b)))`. Затем `eval` вычисляет полученный аргумент и возвращает `b`.

Если необходимо получить динамическое значения для символа, то удобнее использовать функцию `symbol-value`.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

[Переменная] ***evalhook***
[Переменная] ***applyhook***

Если значение `*evalhook*` не является `nil`, тогда `eval` ведёт себя специальным образом. Не-/false значение `*evalhook*` должно быть функцией, которая принимает два аргумента, форму и окружение. Эта функция называется «функцией-ловушкой для вычислителя». Когда форма должна быть вычислена (любая, даже просто число или символ) неявно или явно с помощью `eval`, то вместо вычисления вызывается данная функция с формой в первом аргументе. Тогда функция-ловушка несёт ответственность за вычисление формы, и все что она вернёт будет расценено как результат вычисления этой формы.

Переменная `*applyhook*` похожа на `*evalhook*`, но используется, когда функция должна быть применена к аргументам. Если значение `*applyhook*` не `nil`, тогда `eval` ведёт себя специальным образом.

X3J13 voted in January 1989 to revise the definition of `*applyhook*`. Its value should be a function of *two* arguments, a function and a list of arguments; no environment information is passed to an apply hook function.

This was simply a flaw in the first edition. Sorry about that.

Когда функция должна примениться к списку аргументов, то вызывается функция-ловушка с данной функцией и списком аргументов в качестве параметров. Выполнение формы доверяется функции-ловушке. То, что она вернёт, будет расценено как результат вычисления формы. Функция-ловушка используется для применения обычных функций внутри `eval`. Она не используется для вызовов `apply` или `funcall`, таких функций, как `map` или `reduce`, или вызовов функций раскрытия макросов, таких как `eval` или `macroexpand`.

X3J13 voted in June 1988 to specify that the value of `*macroexpand-hook*` is first coerced to a function before being called as the expansion interface hook. This vote made no mention of `*evalhook*` or `*applyhook*`, but this may have been an oversight.

A proposal was submitted to X3J13 in September 1989 to specify that the value of `*evalhook*` or `*applyhook*` is first coerced to a function before being called. If this proposal is accepted, the value of either variable may be `nil`, any other symbol, a lambda-expression, or any object of type `function`.

Последний аргумент помещаемый в функции-ловушки содержит информацию о лексическом окружении в формате, который зависит от реализации. Эти аргументы одинаковы для функций `evalhook`, `applyhook` и `macroexpand`.

Когда вызывается одна из функций-ловушек, то обе переменные `*evalhook*` и `*applyhook` связываются со значениями `nil` на время выполнения данных функций. Это сделано для того, чтобы функция-ловушка не зациклилась. Функции `evalhook` и `applyhook` полезны для выполнения рекурсивных вычислений и применений (функции) с функцией ловушкой.

Функциональность ловушки предоставляется для облегчения отладки. Функциональность `step` реализована с помощью такой ловушки.

Если случается нелокальный выход на верхний уровень Lisp'a, возможно потому, что ошибка не может быть исправлена, тогда

`*evalhook*` и `*applyhook*` в целях безопасности автоматически сбрасываются в `nil`.

[Функция] **evalhook** *form evalhookfn applyhookfn* **&optional** *env*
[Функция] **applyhook** *function args evalhookfn applyhookfn* **&optional** *env*

Функции `evalhook` и `applyhook` представлены для облегчения использования функциональности ловушек.

В случае `evalhook` вычисляется форма *form*. В случае `applyhook` функция *function* применяется к списку аргументов *args*. В обоих случаях, в процессе выполнения операции переменная `*evalhook*` связана с *evalhookfn*, и `*applyhook*` с *applyhookfn*. Кроме того, аргумент *env* используется для установки лексического окружения. По-умолчанию *env* установлен в нулевое окружение. The check for a hook function is *bypassed* for the evaluation of the *form* itself (for `evalhook`) or for the application of the *function* to the *args* itself (for `applyhook`), but not for subsidiary evaluations and applications such as evaluations of subforms. It is this one-shot bypass that makes `evalhook` and `applyhook` so useful. FIXME

X3J13 voted in January 1989 to eliminate the optional *env* parameter to `applyhook`, because it is not (and cannot) be useful. Any function that can be applied carries its own environment and does not need another environment to be specified separately. This was a flaw in the first edition.

Вот пример, очень простой функции трассировки, которая использует возможности `evalhook`.

```
(defvar *hooklevel* 0)

(defun hook (x)
  (let ((*evalhook* 'eval-hook-function))
    (eval x)))

(defun eval-hook-function (form &rest env)
  (let ((*hooklevel* (+ *hooklevel* 1)))
    (format *trace-output* "~%~V@TForm:  ~S"
            (* *hooklevel* 2) form)
    (let ((values (multiple-value-list
                   (evalhook form
```

```
                        #'eval-hook-function
                        nil
                        env))))
    (format *trace-output* "~%~V@TValue:~{ ~S~}"
            (* *hooklevel* 2) values)
    (values-list values))))
```

Используя этот функционал можно, например, увидеть такую последовательность:

```
(hook '(cons (floor *print-base* 2) 'b))
  Form: (CONS (FLOOR *PRINT-BASE* 2) (QUOTE B))
    Form: (FLOOR *PRINT-BASE* 3)
      Form: *PRINT-BASE*
      Value: 10
      Form: 3
      Value: 3
    Value: 3 1
    Form: (QUOTE B)
    Value: B
  Value: (3 . B)
(3 . B)
```

[Функция] **constantp** *object*

Если предикат `constantp` для объекта *object* истинен, то данный объект, когда рассматривается как вычисляемая форма, всегда вычисляется в одно и то же значение. Константные объекты включают самовычисляемые объекты, такие как числа, строковые символы, строки, битовые вектора, ключевые символы, а также символы констант, определённых с помощью `defconstant`, `nil`, `t` и `pi`. В дополнение, список, у которого *car* элемент равен `quote`, например `(quote foo)`, также является константным объектом.

Если `constantp` для объекта *object* ложен, то этот объект, рассматриваемый как форма, может не всегда вычисляться в одно и то же значение.

20.2 Цикл взаимодействия с пользователем

С Lisp'ом можно работать через специальный цикл вида «`read-eval-print` и сначала». Изменять состояние Lisp системы можно вызовом в этом цикле действий, имеющих побочные эффекты.

Точное определение такого цикла для Common Lisp'а не указывается здесь специально, оставляя разработчикам поле для творчества. Например, они могут сделать командную строку, или простой текстовый редактор, или сложный графический интерфейс. Разработчики могут предоставлять явный запрос ввода, или (как в MacLisp'е) не загромождать экран подсказками.

Цикл взаимодействия с пользователем должен ловить все исключения и изящно их обрабатывать. Он должен также выводить все результаты вычисления форм, возможно в отдельных строках. Если форма вернула ноль значений, это также должно быть отображено.

Следующие переменные управляются циклом взаимодействия с пользователем, для удобства работы, например в случае, если пользователь забыл сохранить интересное введённое выражение или выведенное значение. (Следует отметить, что имена некоторых этих переменных нарушают нотацию глобальных переменных в использовании звёздочек в начале и конце имени.) Эти переменные в основном используются для взаимодействия с пользователем, поэтому имеют короткие имена. Использование этих переменных в программах следует избегать.

[Переменная] +
[Переменная] ++
[Переменная] +++

Когда форма вычисляется в цикле взаимодействия, переменная `+` связывается с предыдущим прочитанным выражением. Переменная `++` хранит предыдущее относительно значения `+` (то есть, форма вычисленная два шага назад), и `+++` хранит предыдущее значение относительно `++`.

[Переменная] **-**

Когда формы выполняется в цикле взаимодействия, переменная `-` связана с этой формой. Это значение, которое получит переменная `+` на

следующей итерации цикла.

[Переменная] *
[Переменная] **
[Переменная] ***

Во время вычисления формы в цикле взаимодействия, переменная * связана с результатом выполнения предыдущей формы, то есть значения формы, которая хранится в +. Если результат той формы содержал несколько значений, в * будет только первое. Если было возвращено ноль значений, * будет содержать `nil`. Переменная ** хранит предыдущее значение относительно *. То есть, результат вычисления формы из **. *** хранит предыдущее значение относительно ***.

Если выполнение + было по каким-то причинам прервано, тогда значения *, ** и *** не меняются. Они изменяются, если вывод значений как минимум начался (необязательно, чтобы он закончился).

[Переменная] /
[Переменная] //
[Переменная] ///

Во время вычисления формы в цикле взаимодействия, переменная / связана со списком результатов выведенных на предыдущей итерации. То есть это список всех значений выполнения формы +. Значение * должно всегда совпадать со значением *car* элемента значения /. Переменная // хранит предыдущий список значений относительно / (то есть, результат вычисленный две итерации назад), и /// содержит предыдущий список значений относительно ///. Таким образом, значение ** должно всегда совпадать со значением *car* элемента списка //, а *** с с *car* элементом ///.

Если выполнение + было по каким-то причинам прервано, тогда значения /, // и /// не меняются. Они изменяются, если вывод значений как минимум начался (необязательно, чтобы он закончился).

В качестве примера работы с этими переменным, рассмотрим следующую возможную работу с системой, где > приглашение ввода:

```
>(cons - -)                ;Итерация 1
((CONS - -) CONS - -)      ;Очаровашка?
```

```
>(values)                ;Итерация 2
                         ;Ничего не выводится
>(cons 'a 'b)            ;Итерация 3
(A . B)                  ;Это одно значение

>(hairy-loop)^G          ;Итерация 4
### QUIT to top level.   ;(Пользователь прервал вычисления.)

>(floor 13 4)            ;Итерация 5
3                        ;Вернулось два значения
1
```

В этой точке мы имеем:

+++ ⇒ (cons 'a 'b)	*** ⇒ NIL	/// ⇒ ()
++ ⇒ (hairy-loop)	** ⇒ (A . B)	// ⇒ ((A . B))
+ ⇒ (floor 13 4)	* ⇒ 3	/ ⇒ (3 1)

Глава 21

Потоки

Потоки является объектами, которые служат в качестве источников или получателей данных. Потоки символов (или символьные потоки) возвращают или принимают строковые символы. Бинарные потоки возвращают или принимают целые числа. Обычное действие Common Lisp системы заключается в чтении символов из символьного входного потока, распознавании символов как представлений Common Lisp'овых объектов данных, вычислении каждого объекта (как формы) и выводе результата в выходной символьный поток.

Обычно потоки соединены с файлами или интерактивными терминалами. Потоки, будучи Lisp'овыми объектами, служат соединителями со внешними устройствами, с помощью которых осуществляется ввод/вывод информации.

Потоки, символьные или бинарные, могут быть только для чтения, только для записи, или для чтения и записи. Какие действия могут производиться над потоком зависит от того, к какому из шести типов он принадлежит.

21.1 Стандартные потоки

В Lisp системе есть несколько переменных, значения которых является потоками, используемыми большим количеством функций. Эти переменные и их использование описаны ниже. По соглашению, переменные, которые содержат поток для чтения, имеют имена заканчивающиеся на `-input`, и переменные, которые содержат поток для

записи, имеют имена, заканчивающиеся на `-output`. Имена переменных, содержащих потоки и для чтения, и для записи, заканчиваются на `-io`.

[Переменная] ***standard-input***

В обычном Lisp'овом цикле взаимодействия с пользователем, входные данные читаются из `*standard-input*` (то есть, из потока, который является значением глобальной переменной `*standard-input*`). Большинство функций, включая `read` и `read-char`, принимают аргумент — поток, который по-умолчанию `*standard-input*`.

[Переменная] ***standard-output***

В обычном Lisp'овом цикле взаимодействия с пользователем, выходные данные посылаются в `*standard-output*` (то есть, в поток, который является значением глобальной переменной `*standard-output*`). Большинство функций, включая `print` и `write-char`, принимают аргумент – поток, который по-умолчанию `*standard-output*`.

[Переменная] ***error-output***

Значение `*error-output*` является потоком, в который должны посылаться сообщения об ошибках. Обычно значение совпадает с `*standard-output*`, но `*standard-output*` может быть связан с файлов и `*error-output*` остаётся направленной на терминал или отдельный файл для сообщений об ошибках.

[Переменная] ***query-io***

Значение `*query-io*` является потоком, используемым, когда необходимо получить от пользователя ответ на некоторый вопрос. Вопрос должен быть выведен в этот поток, и ответ из него прочитан. Когда входной поток для программы может производится из файла, вопрос «Вы действительно хотите удалить все файлы в вашей директории?» никогда не должен посылаться напрямую к пользователю. И ответ должен прийти от пользователя, а не из данных файла. Поэтому в этих целях, вместо `*standard-input*` и `*standard-output*`, должен использоваться `*query-io*` с помощью функции `yes-or-no-p`.

[Переменная] ***debug-io***

Значение `*debug-io*` является потоком, используемым для интерактивной отладки. Часто может совпадать с `*query-io*`, но это необязательно.

[Переменная] ***terminal-io***

Значение `*terminal-io*` является потоком, который соединён с пользовательской консолью. Обычно, запись в этот поток выводит данные на экран, например, а чтение из потока осуществляет чтение ввода с клавиатуры.

Когда стандартные функции, такие как `read` и `read-char` используются с этим потоком, то происходит копирование входных данных обратно в поток или «эхо». (Способ, с помощью которого это происходит, зависит от реализации.)

[Переменная] ***trace-output***

Значение `*trace-output*` является потоком, в который функция `trace` выводит информацию.

Переменные `*standard-input*`, `*standard-output*`, `*error-output*`, `*trace-output*`, `*query-io*` и `*debug-io*` первоначально связаны с потоками-синонимами, которые направляют все операции в поток `*terminal-io*`. (Смотрите `make-synonym-stream`.) Таким образом все проделанные операции на этих потоках отобразятся на терминале.

X3J13 voted in January 1989 to replace the requirements of the preceding paragraph with the following new requirements:

The seven standard stream variables, `*standard-input*`, `*standard-output*`, `*query-io*`, `*debug-io*`, `*terminal-io*`, `*error-output*`, and `*trace-output*`, are initially bound to open streams. (These will be called *the standard initial streams*.)

The streams that are the initial values of `*standard-input*`, `*query-io*`, `*debug-io*`, and `*terminal-io*` must support input.

The streams that are the initial values of `*standard-output*`, `*error-output*`, `*trace-output*`, `*query-io*`, `*debug-io*`, and `*terminal-io*` must support output.

None of the standard initial streams (including the one to which `*terminal-io*` is initially bound) may be a synonym, either directly or indirectly, for any of the standard stream variables except `*terminal-io*`. For example, the initial value of `*trace-output*` may be a synonym stream for `*terminal-io*` but not a synonym stream for `*standard-output*` or `*query-io*`. (These are examples of direct synonyms.) As another example, `*query-io*` may be a two-way stream or echo stream whose input component is a synonym for `*terminal-io*`, but its input component may not be a synonym for `*standard-input*` or `*debug-io*`. (These are examples of indirect synonyms.)

Any or all of the standard initial streams may be direct or indirect synonyms for one or more common implementation-dependent streams. For example, the standard initial streams might all be synonym streams (or two-way or echo streams whose components are synonym streams) to a pair of hidden terminal input and output streams maintained by the implementation.

Part of the intent of these rules is to ensure that it is always safe to bind any standard stream variable to the value of any other standard stream variable (that is, unworkable circularities are avoided) without unduly restricting implementation flexibility.

Пользовательская программа не должна изменять значение `*terminal-io*`. Программа, которая, например, хочет перенаправить вывод в файл, должна изменить значение переменной `*standard-output*`. В таком случае, сообщения об ошибках будут продолжать посылаться в `*error-output*`, а следовательно в `*terminal-io*`, и пользователи сможет их увидеть.

21.2 Создание новых потоков

Пожалуй самые важные конструкции для создания новых потоков это то, которые открывают файлы. Смотрите `with-open-file` и `open`. Следующие функции создают потоки без ссылок на файловую систему.

[Функция] **make-synonym-stream** *symbol*

`make-synonym-stream` создаёт и возвращает поток-синоним. Любые операции на новом потоке будут проделаны на потоке, являющемся

значением динамической переменной с именем *symbol*. Если значение этой переменной изменится или будет пересвязано, то поток-синоним будет воздействовать на новый установленный поток.

The result of `make-synonym-stream` is always a stream of type `synonym-stream`. Note that the type of a synonym stream is *always* `synonym-stream`, regardless of the type of the stream for which it is a synonym.

Результат `make-synonym-stream` всегда является потоком типа `synonym-stream`.

Следует отметить, что тип потока-синонима *всегда* `synonym-stream`, вне зависимости от того, какой тип у связанного потока.

[Функция] **make-broadcast-stream &rest** *streams*

Эта функция возвращает поток, который работает только для записи. Любая выходная информация, посланная в этот поток, будет отослана в все указанные потоки *streams*. Множество операций, которые могут быть выполнены на новом потоке, является пересечением множеств операций для указанных потоков. Результаты, возвращаемые операциями над новым потоком, являются результатами возвращёнными операциями на последнем потоке из списка *streams*. Результаты полученные в ходе выполнения функции над всеми, кроме последнего, потоками игнорируются. Если не было передано ни одного потока в аргументе *streams*, тогда результат является «кусочком клоаки». Вся выводимая информация будет игнорироваться.

Результат `make-broadcast-stream` всегда является потоком типа `broadcast-stream`.

[Функция] **make-concatenated-stream &rest** *streams*

Данная функция возвращает поток, который работает только для чтения. Входная информация берётся из первого потоки из списка *streams* пока указатель не достигнет конца-файла end-of-file, затем данный поток откладывается, и входная информация берётся из следующего, и так далее. Если список потоков *stream* был пуст, то возвращается поток без содержимого. Любая попытка чтения будет возвращать конец-файла end-of-file.

Результат `make-concatenated-stream` всегда является потоком типа `concatenated-stream`.

[Функция] **make-two-way-stream** *input-stream output-stream*

Данная функция возвращает поток для чтения и записи, который входную информацию получает из *input-stream* и посылает выходную информацию в *output-stream*.

Результат `make-two-way-stream` всегда является потоком типа `two-stream-stream`.

[Функция] **make-echo-stream** *input-stream output-stream*

Данная функция возвращает поток для чтения и записи, который получает входную информацию из *input-stream* и отсылает выходную в *output-stream*. В дополнение, входная информация посылается в *output-stream* (эхо).

Результат `make-echo-stream` всегда является потоком типа `echo-stream`.

X3J13 voted in January 1989 to clarify the interaction of `read-char`, `unread-char`, and `peek-char` with echo streams. (See the descriptions of those functions for details.)

X3J13 explicitly noted that the bidirectional streams that are the initial values of `*query-io*`, `*debug-io*`, and `*terminal-io*`, even though they may have some echoing behavior, conceptually are not necessarily the products of calls to `make-echo-stream` and therefore are not subject to the new rules about echoing on echo streams. Instead, these initial interactive streams may have implementation-dependent echoing behavior.

[Функция] **make-string-input-stream** *string* **&optional** *start end*

Данная функция возвращает поток для чтения. Данный поток последовательно будет сохранять строковые символы в подстроке в строке *string* ограниченной с помощью *start* и *end*. После того, как будет достигнут последний символ, поток вернёт конец-файла.

Результат `make-string-input-stream` всегда является потоком типа `string-stream`.

[Функция] **make-string-output-stream** **&key** *:element-type*

Данная функция возвращает поток для записи, который будет аккумулировать всю полученную информацию в строку, которая может быть получена с помощью функции `get-output-stream-string`.

Аргумент `:element-type` указывает, какие символы могут приниматься потоком. Если аргумент `:element-type` опущен, созданный поток должен принимать все символы.

Результатом `make-string-output-stream` всегда является поток типа `string-stream`.

[Функция] **get-output-stream-string** *string-output-stream*

Данная функция возвращает строку, для потока, возвращённого функцией `make-string-output-stream`, которая содержит все записанную в данный поток информацию. После этого поток сбрасывается. Таким образом каждый вызов `get-output-stream-string` возвращает только те символы, которые были записаны с момента предыдущего вызова этой функции (или создания потока, если предыдущего вызова ещё не было).

[Макрос] **with-open-stream** (var stream) {declaration}* {form}*

Форма *stream* вычисляется и должна вернуть поток. Переменная *var* связывается с этим потоком, и затем выполняются формы тела как неявный `progn`. Результатом выполнения `with-open-stream` является значение последней формы. Поток автоматически закрывается при выходе из формы `with-open-stream`, вне зависимости от типа выхода. Смотрите `close`. Поток следует рассматривать, как имеющий динамическую продолжительность видимости.

Результатом `with-open-stream` всегда является поток типа `file-stream`.

[Макрос] **with-input-from-string** (var string {keyword value}*) {declaration}* {form}*

Тело выполняется как неявный `progn` с переменной *var* связанной с потоком символов для чтения, который последовательно предоставляет символы из значения формы *string*. `with-input-from-string` возвращает результат выполнения последней формы *form* тела.

Результатом `with-input-from-stream` всегда является поток типа `string-stream`.

В параметрах могут использоваться следующие имена:

`:index` Форма после `:index` должна быть *местом*, в которое можно осуществить запись с помощью `setf`. Если форма `with-input-from-string` завершается

нормально, то *место* будет содержать позицию первого не прочитанного символа из строки *string* (или длину строки, если все символы были прочитаны). *Место* не изменяется в процессе чтения, а только во время выхода.

`:start` `:start` принимает аргумент, указывающий позицию с которой необходимо начинать чтение символов из строки *string*.

`:end` The `:end` keyword takes an argument indicating, in the manner usual for sequence functions, the end of a substring of *string* to be used. `:end` принимает аргумент, указывающий на позицию на которой необходимо завершить чтение символов из строки *string*

The `:start` and `:index` keywords may both specify the same variable, which is a pointer within the string to be advanced, perhaps repeatedly by some containing loop.

Вот простой пример использования `with-input-from-string`:

```
(with-input-from-string (s "Animal Crackers" :index j :start 6)
  (read s)) ⇒ crackers
```

В качестве побочного эффекта переменная `j` будет установлена в `15`.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

`:start` и `:index` могут оба содержать одну переменную, указывающую позицию в строке, возможно, внутри цикла.

[Макрос] **with-output-to-string** (var [string [`:element-type` type]]) {declaration}* {form}*

Можно указать `nil` вместо строки *string* и использовать аргумент `:element-type` для указания, какие символы должны приниматься созданным потоком. Если аргумент *string* не указан или он `nil` и не указан `:element-type`, то созданный поток должен принимать все символы.

If *string* is specified, it must be a string with a fill pointer; the output is incrementally appended to the string (as if by use of `vector-push-extend`).

In this way output cannot be accidentally lost. This change makes `with-output-to-string` behave in the same way that `format` does when given a string as its first argument.

Результатом `with-output-to-stream` всегда является поток типа `string-stream`.

X3J13 voted in January 1989 to restrict user side effects; see section 7.9.

21.3 Операции над потоками

В этом разделе описаны только те функции, которые работают со всеми потоки. Ввод и вывод информации слегка сложнее и описаны отдельно в главе 22. Интерфейс между потоками и файловой системой описан в главе 23

[Функция] **streamp** *object*

`streamp` истинен, если его аргумент является потоком, иначе ложен.

(streamp x) ≡ (typep x 'stream)

`streamp` is unaffected by whether its argument, if a stream, is open or closed. In either case it returns true.

[Функция] **open-stream-p** *stream*

Если аргумент, который должен быть потоком, открыт, предикат истинен, иначе ложен.

Поток всегда создаётся открытым. Он продолжает быть открытым, пока не будет закрыт с помощью функции `close`. Макросы `with-open-stream`, `with-input-from-string`, `with-output-to-string` и `with-open-file` автоматически закрывают созданный поток, когда управление выходит из их тел, по сути открытость совпадает с динамической продолжительностью видимости потока.

[Функция] **input-stream-p** *stream*

Если аргумент, который должен быть потоком, может работать для чтения, предикат истинен, иначе ложен.

[Функция] **output-stream-p** *stream*

Если аргумент, который должен быть потоком, может работать для записи, предикат истинен, иначе ложен.

[Функция] **stream-element-type** *stream*

Функция возвращает спецификатор типа, который указывает на то, какие объекты могут быть прочитаны или записаны из/в поток *stream*. Потоки созданные с помощью `open` будут иметь тип элементов, ограниченный подмножеством `character` или `integer`. Но в принципе поток может проводить операции используя любые Lisp'овые объекты.

[Функция] **close** *stream* **&key** *:abort*

Аргумент должен быть потоком. Функцией этот поток закрывается. После чего операции чтения и записи выполняться над ним не могут. Однако, конечно, некоторые операции все ещё могут выполняться. Допускается повторное закрытие уже закрытого потока.

X3J13 voted in January 1989 and revised the vote in March 1989 to specify that if `close` is called on an open stream, the stream is closed and `t` is returned; but if `close` is called on a closed stream, it succeeds without error and returns an unspecified value. (The rationale for not specifying the value returned for a closed stream is that in some implementations closing certain streams does not really have an effect on them—for example, closing the `*terminal-io*` stream might not "really" close it—and it is not desirable to force such implementations to keep otherwise unnecessary state. Portable programs will of course not rely on such behavior.)

X3J13 also voted in January 1989 to specify exactly which inquiry functions may be applied to closed streams:

`streamp`	`pathname-host`	`namestring`
`pathname`	`pathname-device`	`file-namestring`
`truename`	`pathname-directory`	`directory-namestring`
`merge-pathnames`	`pathname-name`	`host-namestring`
`open`	`pathname-type`	`enough-namestring`
`probe-file`	`pathname-version`	`directory`

See the individual descriptions of these functions for more information on how they operate on closed streams.

X3J13 voted in January 1989 to clarify the effect of closing various kinds of streams. First some terminology:

- A *composite* stream is one that was returned by a call to `make-synonym-stream`, `make-broadcast-stream`, `make-concatenated-stream`, `make-two-way-stream`, or `make-echo-stream`.

- The *constituents* of a composite stream are the streams that were given as arguments to the function that constructed it or, in the case of `make-synonym-stream`, the stream that is the `symbol-value` of the symbol that was given as an argument. (The constituent of a synonym stream may therefore vary over time.)

- A *constructed* stream is either a composite stream or one returned by a call to `make-string-input-stream`, `make-string-output-stream`, `with-input-from-string`, or `with-output-to-string`.

The effect of applying `close` to a constructed stream is to close that stream only. No input/output operations are permitted on the constructed stream once it has been closed (though certain inquiry functions are still permitted, as described above).

Closing a composite stream has no effect on its constituents; any constituents that are open remain open.

If a stream created by `make-string-output-stream` is closed, the result of then applying `get-output-stream-string` to the stream is unspecified.

Если параметр `:abort` не-`nil` (а по-умолчанию он `nil`), то он указывает на ненормальное завершение использования потока. Осуществляется попытка убрать все побочные эффекты, созданные потоком. Например, если поток выполнял вывод в файл, который был создан вместе с потоком, тогда, если возможно, файл удаляется и любой ранее существовавший файл не заменяется.

[Функция] **broadcast-stream-streams** *broadcast-stream*

Аргумент должен быть типа `broadcast-stream`. Функцией возвращается список потоков для записи (и открытых, и нет).

[Функция] **concatenated-stream-streams** *concatenated-stream*

Аргумент должен быть типа `concatenated-stream`. Функцией возвращается список потоков (и открытых, и нет). Этот список отображает упорядоченное множество потоков для чтения, из которых поток *concatenated-stream* все ещё может получать данные. Поток, из которого в данный момент читались данные, находится в начале списка. Если потоков для чтения нет, список может быть пустым.

[Функция] **echo-stream-input-stream** *echo-stream*
[Функция] **echo-stream-output-stream** *echo-stream*

Аргумент должен быть типа `echo-stream`. Функция `echo-stream-input-stream` возвращает список потоков для чтения. `echo-stream-output-stream` возвращает список потоков для записи.

[Функция] **synonym-stream-symbol** *synonym-stream*

Аргумент должен быть типа `synonym-stream`. Эта функция возвращает символ, значение которого является потоком для потока-синонима *synonym-stream*.

[Функция] **two-way-stream-input-stream** *two-way-stream*
[Функция] **two-way-stream-output-stream** *two-way-stream*

Аргумент должен быть типа `two-way-stream`. Функция `two-way-stream-input-stream` возвращает список потоков для чтения. `two-way-stream-output-stream` возвращает список потоков для записи.

[Function] **interactive-stream-p** *stream*

X3J13 voted in June 1989 to add the predicate `interactive-stream-p`, which returns `t` if the *stream* is interactive and otherwise returns `nil`. A `type-error` error is signalled if the argument is not of type `stream`.

The precise meaning of `interactive-stream-p` is implementation-dependent and may depend on the underlying operating system. The intent is to distinguish between interactive and batch (background, command-file) operations. Some characteristics that might distinguish a stream as interactive:

- The stream is connected to a person (or the equivalent) in such a way that the program can prompt for information and expect to receive input that might depend on the prompt.
- The program is expected to prompt for input and to support "normal input editing protocol" for that operating environment.
- A call to `read-char` might hang waiting for the user to type something rather than quickly returning a character or an end-of-file indication.

The value of `*terminal-io*` might or might not be interactive.

[Function] **stream-external-format** *stream*

X3J13 voted in June 1989 to add the function `stream-external-format`, which returns a specifier for the implementation-recognized scheme used for representing characters in the argument *stream*. See the `:external-format` argument to `open`.

Глава 22

Input/Output Ввод/Вывод

Common Lisp provides a rich set of facilities for performing input/output. All input/output operations are performed on streams of various kinds. This chapter is devoted to stream data transfer operations. Streams are discussed in chapter 21, and ways of manipulating files through streams are discussed in chapter 23.

While there is provision for reading and writing binary data, most of the I/O operations in Common Lisp read or write characters. There are simple primitives for reading and writing single characters or lines of data. The `format` function can perform complex formatting of output data, directed by a control string in manner similar to a Fortran `FORMAT` statement or a `PL/I PUT EDIT` statement. The most useful I/O operations, however, read and write printed representations of arbitrary Lisp objects.

Common Lisp содержит богатый функционал для выполнения операций ввода/вывода. Все эти операции производятся на различного вида потоках. Данная глава посвящена тому, как оперировать данными в потоках. Потоки обсуждаются в главе 21, а способы работы с файлами через потоки в главе 23.

Большинство операций ввода/вывода в Common Lisp'е читают и записывают строковые символы, но также есть функции и для бинарных данных. Есть простые примитивы для чтения и записи одного символа или строк данных. Функция `format` функции может выполнять сложное форматирование выходных данных, направленных управляющей строкой как в выражении `FORMAT` в Fortran'е или в `PUT EDIT` в `PL/I`. Однако, самые полезные операции ввода/вывода читают и записывает выводимые представления произвольных Lisp'овых

объектов.

22.1 Printed Representation of Lisp Objects

Lisp objects in general are not text strings but complex data structures. They have very different properties from text strings as a consequence of their internal representation. However, to make it possible to get at and talk about Lisp objects, Lisp provides a representation of most objects in the form of printed text; this is called the *printed representation*, which is used for input/output purposes and in the examples throughout this book. Functions such as `print` take a Lisp object and send the characters of its printed representation to a stream. The collection of routines that does this is known as the (Lisp) *printer*. The `read` function takes characters from a stream, interprets them as a printed representation of a Lisp object, builds that object, and returns it; the collection of routines that does this is called the (Lisp) *reader*.

Ideally, one could print a Lisp object and then read the printed representation back in, and so obtain the same identical object. In practice this is difficult and for some purposes not even desirable. Instead, reading a printed representation produces an object that is (with obscure technical exceptions) `equal` to the originally printed object.

Most Lisp objects have more than one possible printed representation. For example, the integer twenty-seven can be written in any of these ways:

```
27    27.    #o33    #x1B    #b11011    #.(* 3 3 3)    81/3
```

A list of two symbols `A` and `B` can be printed in many ways:

```
(A B)    (a b)    (  a  b )    (\A |B|)
(|\A|
  B
)
```

The last example, which is spread over three lines, may be ugly, but it is legitimate. In general, wherever whitespace is permissible in a printed representation, any number of spaces and newlines may appear.

When `print` produces a printed representation, it must choose arbitrarily from among many possible printed representations. It attempts to choose one that is readable. There are a number of global variables that can be used to control the actions of `print`, and a number of different printing functions.

This section describes in detail what is the standard printed representation for any Lisp object and also describes how `read` operates.

22.1.1 What the Read Function Accepts

The purpose of the Lisp reader is to accept characters, interpret them as the printed representation of a Lisp object, and construct and return such an object. The reader cannot accept everything that the printer produces; for example, the printed representations of compiled code objects cannot be read in. However, the reader has many features that are not used by the output of the printer at all, such as comments, alternative representations, and convenient abbreviations for frequently used but unwieldy constructs. The reader is also parameterized in such a way that it can be used as a lexical analyzer for a more general user-written parser.

The reader is organized as a recursive-descent parser. Broadly speaking, the reader operates by reading a character from the input stream and treating it in one of three ways. Whitespace characters serve as separators but are otherwise ignored. Constituent and escape characters are accumulated to make a *token*, which is then interpreted as a number or symbol. Macro characters trigger the invocation of functions (possibly user-supplied) that can perform arbitrary parsing actions, including recursive invocation of the reader.

More precisely, when the reader is invoked, it reads a single character from the input stream and dispatches according to the syntactic type of that character. Every character that can appear in the input stream must be of exactly one of the following kinds: *illegal*, *whitespace*, *constituent*, *single escape*, *multiple escape*, or *macro*. Macro characters are further divided into the types *terminating* and *non-terminating* (of tokens). (Note that macro characters have nothing whatever to do with macros in their operation. There is a superficial similarity in that macros allow the user to extend the syntax of Common Lisp at the level of forms, while macro characters allow the user to extend the syntax at the level of characters.) Constituents additionally have one or more attributes, the most important of which is *alphabetic*; these

attributes are discussed further in section 22.1.2.

The parsing of Common Lisp expressions is discussed in terms of these syntactic character types because the types of individual characters are not fixed but may be altered by the user (see `set-syntax-from-char` and `set-macro-character`). The characters of the standard character set initially have the syntactic types shown in table 22.2. Note that the brackets, braces, question mark, and exclamation point (that is, `[`, `]`, `{`, `}`, `?`, and `!`) are normally defined to be constituents, but they are not used for any purpose in standard Common Lisp syntax and do not occur in the names of built-in Common Lisp functions or variables. These characters are explicitly reserved to the user. The primary intent is that they be used as macro characters; but a user might choose, for example, to make `!` be a *single escape* character (as it is in Portable Standard Lisp).

The algorithm performed by the Common Lisp reader is roughly as follows:

1. If at end of file, perform end-of-file processing (as specified by the caller of the `read` function). Otherwise, read one character from the input stream, call it x, and dispatch according to the syntactic type of x to one of steps 2 to 7.

2. If x is an *illegal* character, signal an error.

3. If x is a *whitespace* character, then discard it and go back to step 1.

4. If x is a *macro* character (at this point the distinction between *terminating* and *non-terminating* macro characters does not matter), then execute the function associated with that character. The function may return zero values or one value (see `values`).

   The macro-character function may of course read characters from the input stream; if it does, it will see those characters following the macro character. The function may even invoke the reader recursively. This is how the macro character `(` constructs a list: by invoking the reader recursively to read the elements of the list.

   If one value is returned, then return that value as the result of the read operation; the algorithm is done. If zero values are returned, then go back to step 1.

Таблица 22.1: Standard Character Syntax Types

⟨tab⟩	*whitespace*	⟨page⟩	*whitespace*	⟨newline⟩	*whitespace*
⟨space⟩	*whitespace*	@	*constituent*	‘	*terminating macro*
!	*constituent* *	A	*constituent*	a	*constituent*
"	*terminating macro*	B	*constituent*	b	*constituent*
#	*non-terminating macro*	C	*constituent*	c	*constituent*
$	*constituent*	D	*constituent*	d	*constituent*
%	*constituent*	E	*constituent*	e	*constituent*
&	*constituent*	F	*constituent*	f	*constituent*
’	*terminating macro*	G	*constituent*	g	*constituent*
(	*terminating macro*	H	*constituent*	h	*constituent*
)	*terminating macro*	I	*constituent*	i	*constituent*
*	*constituent*	J	*constituent*	j	*constituent*
+	*constituent*	K	*constituent*	k	*constituent*
,	*terminating macro*	L	*constituent*	l	*constituent*
-	*constituent*	M	*constituent*	m	*constituent*
.	*constituent*	N	*constituent*	n	*constituent*
/	*constituent*	O	*constituent*	o	*constituent*
0	*constituent*	P	*constituent*	p	*constituent*
1	*constituent*	Q	*constituent*	q	*constituent*
2	*constituent*	R	*constituent*	r	*constituent*
3	*constituent*	S	*constituent*	s	*constituent*
4	*constituent*	T	*constituent*	t	*constituent*
5	*constituent*	U	*constituent*	u	*constituent*
6	*constituent*	V	*constituent*	v	*constituent*
7	*constituent*	W	*constituent*	w	*constituent*
8	*constituent*	X	*constituent*	x	*constituent*
9	*constituent*	Y	*constituent*	y	*constituent*
:	*constituent*	Z	*constituent*	z	*constituent*
;	*terminating macro*	[	*constituent* *	{	*constituent* *
<	*constituent*	\	*single escape*	|	*multiple escape*
=	*constituent*	]	*constituent* *	}	*constituent* *
>	*constituent*	^	*constituent*	~	*constituent*
?	*constituent* *	_	*constituent*	⟨rubout⟩	*constituent*
⟨backspace⟩	*constituent*	⟨return⟩	*whitespace*	⟨linefeed⟩	*whitespace*

The characters marked with an asterisk are initially constituents but are reserved to the user for use as macro characters or for any other desired purpose.

Таблица 22.2: Стандартные типы символьного синтакса

⟨tab⟩	*пробел*	⟨page⟩	*пробел*	⟨newline⟩	*пробел*
⟨space⟩	*пробел*	@	*составная часть*	‘	*терминальный макрос*
!	*составная часть* *	A	*составная часть*	a	*составная часть*
"	*терминальный макрос*	B	*составная часть*	b	*составная часть*
#	*не-терминальный макрос*	C	*составная часть*	c	*составная часть*
$	*составная часть*	D	*составная часть*	d	*составная часть*
%	*составная часть*	E	*составная часть*	e	*составная часть*
&	*составная часть*	F	*составная часть*	f	*составная часть*
’	*терминальный макрос*	G	*составная часть*	g	*составная часть*
(	*терминальный макрос*	H	*составная часть*	h	*составная часть*
)	*терминальный макрос*	I	*составная часть*	i	*составная часть*
*	*составная часть*	J	*составная часть*	j	*составная часть*
+	*составная часть*	K	*составная часть*	k	*составная часть*
,	*терминальный макрос*	L	*составная часть*	l	*составная часть*
-	*составная часть*	M	*составная часть*	m	*составная часть*
.	*составная часть*	N	*составная часть*	n	*составная часть*
/	*составная часть*	O	*составная часть*	o	*составная часть*
0	*составная часть*	P	*составная часть*	p	*составная часть*
1	*составная часть*	Q	*составная часть*	q	*составная часть*
2	*составная часть*	R	*составная часть*	r	*составная часть*
3	*составная часть*	S	*составная часть*	s	*составная часть*
4	*составная часть*	T	*составная часть*	t	*составная часть*
5	*составная часть*	U	*составная часть*	u	*составная часть*
6	*составная часть*	V	*составная часть*	v	*составная часть*
7	*составная часть*	W	*составная часть*	w	*составная часть*
8	*составная часть*	X	*составная часть*	x	*составная часть*
9	*составная часть*	Y	*составная часть*	y	*составная часть*
:	*составная часть*	Z	*составная часть*	z	*составная часть*
;	*терминальный макрос*	[	*составная часть* *	{	*составная часть* *
<	*составная часть*	\	*экранирующий один*	|	*экранирующий много*
=	*составная часть*	]	*составная часть* *	}	*составная часть* *
>	*составная часть*	^	*составная часть*	~	*составная часть*
?	*составная часть* *	_	*составная часть*	⟨rubout⟩	*составная часть*
⟨backspace⟩	*составная часть*	⟨return⟩	*пробел*	⟨linefeed⟩	*пробел*

Символы помеченный звездочкой первоначально являются составной частью, но зарезервированы для пользователя в качестве использования макросимволов или для других целей.

5. If x is a *single escape* character (normally \), then read the next character and call it y (but if at end of file, signal an error instead). Ignore the usual syntax of y and pretend it is a *constituent* whose only attribute is *alphabetic*. (If y is a lowercase character, leave it alone; do not replace it with the corresponding uppercase character.) For the purposes of `readtable-case`, y is not replaceable. Use y to begin a token, and go to step 8.

6. If x is a *multiple escape* character (normally |), then begin a token (initially containing no characters) and go to step 9.

7. If x is a *constituent* character, then it begins an extended token. After the entire token is read in, it will be interpreted either as representing a Lisp object such as a symbol or number (in which case that object is returned as the result of the read operation), or as being of illegal syntax (in which case an error is signaled). If x is a lowercase character, replace it with the corresponding uppercase character. X3J13 voted in June 1989 to introduce `readtable-case`. Consequently, the preceding sentence should be ignored. The case of x should not be altered; instead, x should be regarded as replaceable. Use x to begin a token, and go on to step 8.

8. (At this point a token is being accumulated, and an even number of *multiple escape* characters have been encountered.) If at end of file, go to step 10. Otherwise, read a character (call it y), and perform one of the following actions according to its syntactic type:

   - If y is a *constituent* or *non-terminating macro*, then do the following. If y is a lowercase character, replace it with the corresponding uppercase character. X3J13 voted in June 1989 to introduce `readtable-case`. Consequently, the preceding sentence should be ignored. The case of y should not be altered; instead, y should be regarded as replaceable. Append y to the token being built, and repeat step 8.
   - If y is a *single escape* character, then read the next character and call it z (but if at end of file, signal an error instead). Ignore the usual syntax of z and pretend it is a *constituent* whose only attribute is *alphabetic*. (If z is a lowercase character, leave it alone; do not replace it with the corresponding uppercase character.) For the

purposes of `readtable-case`, z is not replaceable. Append z to the token being built, and repeat step 8.

- If y is a *multiple escape* character, then go to step 9.
- If y is an *illegal* character, signal an error.
- If y is a *terminating macro* character, it terminates the token. First "unread" the character y (see `unread-char`), then go to step 10.
- If y is a *whitespace* character, it terminates the token. First "unread" y if appropriate (see `read-preserving-whitespace`), then go to step 10.

9. (At this point a token is being accumulated, and an odd number of *multiple escape* characters have been encountered.) If at end of file, signal an error. Otherwise, read a character (call it y), and perform one of the following actions according to its syntactic type:

   - If y is a *constituent*, *macro*, or *whitespace* character, then ignore the usual syntax of that character and pretend it is a *constituent* whose only attribute is *alphabetic*. (If y is a lowercase character, leave it alone; do not replace it with the corresponding uppercase character.) For the purposes of `readtable-case`, y is not replaceable. Append y to the token being built, and repeat step 9.
   - If y is a *single escape* character, then read the next character and call it z (but if at end of file, signal an error instead). Ignore the usual syntax of z and pretend it is a *constituent* whose only attribute is *alphabetic*. (If z is a lowercase character, leave it alone; do not replace it with the corresponding uppercase character.) For the purposes of `readtable-case`, z is not replaceable. Append z to the token being built, and repeat step 9.
   - If y is a *multiple escape* character, then go to step 8.
   - If y is an *illegal* character, signal an error.

10. An entire token has been accumulated. X3J13 voted in June 1989 to introduce `readtable-case`. If the accumulated token is to be interpreted as a symbol, any case conversion of replaceable characters should be performed at this point according to the value of the `readtable-case` slot of the current readtable (the value of `*readtable*`). Interpret the token as

representing a Lisp object and return that object as the result of the read operation, or signal an error if the token is not of legal syntax. X3J13 voted in March 1989 to specify that implementation-defined attributes may be removed from the characters of a symbol token when constructing the print name. It is implementation-dependent which attributes are removed.

As a rule, a *single escape* character never stands for itself but always serves to cause the following character to be treated as a simple alphabetic character. A *single escape* character can be included in a token only if preceded by another *single escape* character.

A *multiple escape* character also never stands for itself. The characters between a pair of *multiple escape* characters are all treated as simple alphabetic characters, except that *single escape* and *multiple escape* characters must nevertheless be preceded by a *single escape* character to be included.

22.1.2 Parsing of Numbers and Symbols

When an extended token is read, it is interpreted as a number or symbol. In general, the token is interpreted as a number if it satisfies the syntax for numbers specified in table 22.4; this is discussed in more detail below.

The characters of the extended token may serve various syntactic functions as shown in table 22.6, but it must be remembered that any character included in a token under the control of an escape character is treated as *alphabetic* rather than according to the attributes shown in the table. One consequence of this rule is that a whitespace, macro, or escape character will always be treated as alphabetic within an extended token because such a character cannot be included in an extended token except under the control of an escape character.

To allow for extensions to the syntax of numbers, a syntax for *potential numbers* is defined in Common Lisp that is more general than the actual syntax for numbers. Any token that is not a potential number and does not consist entirely of dots will always be taken to be a symbol, now and in the future; programs may rely on this fact. Any token that is a potential number but does not fit the actual number syntax defined below is a *reserved token* and has an implementation-dependent interpretation; an implementation may signal an error, quietly treat the token as a symbol, or take some

Таблица 22.3: Actual Syntax of Numbers

number ::= *integer* | *ratio* | *floating-point-number*
integer ::= [sign] {digit}+ [decimal-point]
ratio ::= [sign] {digit}+ `/` {digit}+
floating-point-number ::= [sign] {digit}* *decimal-point* {digit}+ [exponent]
| [sign] {digit}+ [decimal-point {digit}*] *exponent*
sign ::= `+` | `-`
decimal-point ::= `.`
digit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
exponent ::= *exponent-marker* [sign] {digit}+
exponent-marker ::= `e` | `s` | `f` | `d` | `l` | `E` | `S` | `F` | `D` | `L`

other action. Programmers should avoid the use of such reserved tokens. (A symbol whose name looks like a reserved token can always be written using one or more escape characters.)

Just as *bignum* is the standard term used by Lisp implementors for very large integers, and *flonum* (rhymes with "low hum") refers to a floating-point number, the term *potnum* has been used widely as an abbreviation for "potential number." "Potnum" rhymes with "hot rum."

A token is a potential number if it satisfies the following requirements:

- It consists entirely of digits, signs (`+` or `-`), ratio markers (`/`), decimal points (`.`), extension characters (`^` or `_`), and number markers. (A number marker is a letter. Whether a letter may be treated as a number marker depends on context, but no letter that is adjacent to another letter may ever be treated as a number marker. Floating-point exponent markers are instances of number markers.)

- It contains at least one digit. (Letters may be considered to be digits, depending on the value of `*read-base*`, but only in tokens containing no decimal points.)

- It begins with a digit, sign, decimal point, or extension character.

- It does not end with a sign.

Таблица 22.4: Синтаксис чисел

number ::= *integer* | *ratio* | *floating-point-number*
integer ::= [sign] {digit}+ [decimal-point]
ratio ::= [sign] {digit}+ `/` {digit}+
floating-point-number ::= [sign] {digit}* *decimal-point* {digit}+ [exponent]
| [sign] {digit}+ [decimal-point {digit}*] *exponent*
sign ::= `+` | `-`
decimal-point ::= `.`
digit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
exponent ::= *exponent-marker* [sign] {digit}+
exponent-marker ::= `e` | `s` | `f` | `d` | `l` | `E` | `S` | `F` | `D` | `L`

As examples, the following tokens are potential numbers, but they are *not* actually numbers as defined below, and so are reserved tokens. (They do indicate some interesting possibilities for future extensions.)

1b5000	777777q	1.7J	-3/4+6.7J	12/25/83
27^19	3^4/5	6//7	3.1.2.6	^-43^
3.141_592_653_589_793_238_4			-3.7+2.6i-6.17j+19.6k	

The following tokens are *not* potential numbers but are always treated as symbols:

/	/5	+	1+	1-
foo+	ab.cd	_	^	^/-

The following tokens are potential numbers if the value of `*read-base*` is `16` (an abnormal situation), but they are always treated as symbols if the value of `*read-base*` is `10` (the usual value):

bad-face	25-dec-83	a/b	fad_cafe	f^

Таблица 22.5: Standard Constituent Character Attributes

! *alphabetic*	⟨page⟩	*illegal*	⟨backspace⟩	*illegal*	
" *alphabetic* *	⟨return⟩	*illegal* *	⟨tab⟩	*illegal* *	
# *alphabetic* *	⟨space⟩	*illegal* *	⟨newline⟩	*illegal* *	
$ *alphabetic*	⟨rubout⟩	*illegal*	⟨linefeed⟩	*illegal* *	
% *alphabetic*	.	*alphabetic*, *dot*, *decimal point*			
& *alphabetic*	+	*alphabetic*, *plus sign*			
' *alphabetic* *	-	*alphabetic*, *minus sign*			
(*alphabetic* *	*	*alphabetic*			
) *alphabetic* *	/	*alphabetic*, *ratio marker*			
, *alphabetic* *	@	*alphabetic*			
0 *alphadigit*	A, a	*alphadigit*			
1 *alphadigit*	B, b	*alphadigit*			
2 *alphadigit*	C, c	*alphadigit*			
3 *alphadigit*	D, d	*alphadigit*, *double-float exponent marker*			
4 *alphadigit*	E, e	*alphadigit*, *float exponent marker*			
5 *alphadigit*	F, f	*alphadigit*, *single-float exponent marker*			
6 *alphadigit*	G, g	*alphadigit*			
7 *alphadigit*	H, h	*alphadigit*			
8 *alphadigit*	I, i	*alphadigit*			
9 *alphadigit*	J, j	*alphadigit*			
: *package marker*	K, k	*alphadigit*			
; *alphabetic* *	L, l	*alphadigit*, *long-float exponent marker*			
< *alphabetic*	M, m	*alphadigit*			
= *alphabetic*	N, n	*alphadigit*			
> *alphabetic*	O, o	*alphadigit*			
? *alphabetic*	P, p	*alphadigit*			
[*alphabetic*	Q, q	*alphadigit*			
\ *alphabetic* *	R, r	*alphadigit*			
] *alphabetic*	S, s	*alphadigit*, *short-float exponent marker*			
^ *alphabetic*	T, t	*alphadigit*			
_ *alphabetic*	U, u	*alphadigit*			
` *alphabetic* *	V, v	*alphadigit*			
{ *alphabetic*	W, w	*alphadigit*			
| *alphabetic* *	X, x	*alphadigit*			
} *alphabetic*	Y, y	*alphadigit*			
~ *alphabetic*	Z, z	*alphadigit*			

These interpretations apply only to characters whose syntactic type is *constituent*. Entries marked with an asterisk are normally shadowed because the characters are of syntactic type *whitespace*, *macro*, *single escape*, or *multiple escape*. An *alphadigit* character is interpreted as a digit if it is a valid digit in the radix specified by `*read-base*`; otherwise it is alphabetic. Characters with an *illegal* attribute can never appear in a token except under the control of an escape character.

Таблица 22.6: Свойства стандартных символов

!*алфавитный*	⟨page⟩	*недопустимый*	⟨backspace⟩	*недопустимый*	
"*алфавитный* *	⟨return⟩	*недопустимый* *	⟨tab⟩	*недопустимый* *	
#*алфавитный* *	⟨space⟩	*недопустимый* *	⟨newline⟩	*недопустимый* *	
$*алфавитный*	⟨rubout⟩	*недопустимый*	⟨linefeed⟩	*недопустимый* *	
%*алфавитный*	.	*алфавитный*, *точка*, *разделитель десятичной части*			
&*алфавитный*	+	*алфавитный*, *знак плюс*			
'*алфавитный* *	-	*алфавитный*, *знак минус*			
(*алфавитный* *	*	*алфавитный*			
)*алфавитный* *	/	*алфавитный*, *маркер дроби*			
,*алфавитный* *	@	*алфавитный*			
0*алфавитно-цифровой*	A, a	*алфавитно-цифровой*			
1*алфавитно-цифровой*	B, b	*алфавитно-цифровой*			
2*алфавитно-цифровой*	C, c	*алфавитно-цифровой*			
3*алфавитно-цифровой*	D, d	*алфавитно-цифровой*, *маркер экспоненты для двойного с плав*			
4*алфавитно-цифровой*	E, e	*алфавитно-цифровой*, *маркер экспоненты для числа с плаваю*			
5*алфавитно-цифровой*	F, f	*алфавитно-цифровой*, *маркер экспоненты для одинарного с п*			
6*алфавитно-цифровой*	G, g	*алфавитно-цифровой*			
7*алфавитно-цифровой*	H, h	*алфавитно-цифровой*			
8*алфавитно-цифровой*	I, i	*алфавитно-цифровой*			
9*алфавитно-цифровой*	J, j	*алфавитно-цифровой*			
:*package marker*	K, k	*алфавитно-цифровой*			
;*алфавитный* *	L, l	*алфавитно-цифровой*, *маркер экспоненты для длинного с пла*			
<*алфавитный*	M, m	*алфавитно-цифровой*			
=*алфавитный*	N, n	*алфавитно-цифровой*			
>*алфавитный*	O, o	*алфавитно-цифровой*			
?*алфавитный*	P, p	*алфавитно-цифровой*			
[*алфавитный*	Q, q	*алфавитно-цифровой*			
алфавитный *	R, r	*алфавитно-цифровой*			
]*алфавитный*	S, s	*алфавитно-цифровой*, *маркер экспоненты для короткого с пл*			
^*алфавитный*	T, t	*алфавитно-цифровой*			
_*алфавитный*	U, u	*алфавитно-цифровой*			
`*алфавитный* *	V, v	*алфавитно-цифровой*			
{*алфавитный*	W, w	*алфавитно-цифровой*			
|*алфавитный* *	X, x	*алфавитно-цифровой*			
}*алфавитный*	Y, y	*алфавитно-цифровой*			
~*алфавитный*	Z, z	*алфавитно-цифровой*			

These interpretations apply only to characters whose syntactic type is *constituent*. Entries marked with an asterisk are normally shadowed because the characters are of syntactic type *whitespace*, *macro*, *single escape*, or *multiple escape*. An *alphadigit* character is interpreted as a digit if it is a valid digit in the radix specified by `*read-base*`; otherwise it is alphabetic. Characters with an *illegal* attribute can never appear in a token except under the control of an escape character.

It is possible for there to be an ambiguity as to whether a letter should be treated as a digit or as a number marker. In such a case, the letter is always treated as a digit rather than as a number marker.

Note that the printed representation for a potential number may not contain any escape characters. An escape character robs the following character of all syntactic qualities, forcing it to be strictly alphabetic and therefore unsuitable for use in a potential number. For example, all of the following representations are interpreted as symbols, not numbers:

\256 25\64 1.0\E6 |100| 3\.14159 |3/4| 3\/4 5||

In each case, removing the escape character(s) would allow the token to be treated as a number.

If a potential number can in fact be interpreted as a number according to the BNF syntax in table 22.4, then a number object of the appropriate type is constructed and returned. It should be noted that in a given implementation it may be that not all tokens conforming to the actual syntax for numbers can actually be converted into number objects. For example, specifying too large or too small an exponent for a floating-point number may make the number impossible to represent in the implementation. Similarly, a ratio with denominator zero (such as `-35/000`) cannot be represented in *any* implementation. In any such circumstance where a token with the syntax of a number cannot be converted to an internal number object, an error is signaled. (On the other hand, an error must not be signaled for specifying too many significant digits for a floating-point number; an appropriately truncated or rounded value should be produced.)

There is an omission in the syntax of numbers as described in table 22.4, in that the syntax does not account for the possible use of letters as digits. The radix used for reading integers and ratios is normally decimal. However, this radix is actually determined by the value of the variable `*read-base*`, whose initial value is `10`. `*read-base*` may take on any integral value between `2` and `36`; let this value be n. Then a token x is interpreted as an integer or ratio in base n if it could be properly so interpreted in the syntax `#`n`R`x (see section 22.1.4). So, for example, if the value of `*read-base*` is `16`, then the printed representation

(a small face in a bad place)

would be interpreted as if the following representation had been read with `*read-base*` set to `10`:

(10 small 64206 in 10 2989 place)

because four of the seven tokens in the list can be interpreted as hexadecimal numbers. This facility is intended to be used in reading files of data that for some reason contain numbers not in decimal radix; it may also be used for reading programs written in Lisp dialects (such as MacLisp) whose default number radix is not decimal. Non-decimal constants in Common Lisp programs or portable Common Lisp data files should be written using `#O`, `#X`, `#B`, or `#`*n*`R` syntax.

When `*read-base*` has a value greater than `10`, an ambiguity is introduced into the actual syntax for numbers because a letter can serve as either a digit or an exponent marker; a simple example is `1E0` when the value of `*read-base*` is `16`. The ambiguity is resolved in accordance with the general principle that interpretation as a digit is preferred to interpretation as a number marker. The consequence in this case is that if a token can be interpreted as either an integer or a floating-point number, then it is taken to be an integer.

If a token consists solely of dots (with no escape characters), then an error is signaled, except in one circumstance: if the token is a single dot and occurs in a situation appropriate to "dotted list" syntax, then it is accepted as a part of such syntax. Signaling an error catches not only misplaced dots in dotted list syntax but also lists that were truncated by `*print-length*` cutoff, because such lists end with a three-dot sequence (`...`). Examples:

```
(a . b)      ;A dotted pair of a and b
(a.b)        ;A list of one element, the symbol named a.b
(a. b)       ;A list of two elements a. and b
(a .b)       ;A list of two elements a and .b
(a \. b)     ;A list of three elements a, ., and b
(a |.| b)    ;A list of three elements a, ., and b
(a \... b)   ;A list of three elements a, ..., and b
(a |...| b)  ;A list of three elements a, ..., and b
(a b . c)    ;A dotted list of a and b with c at the end
.iot         ;The symbol whose name is .iot
(. b)        ;Illegal; an error is signaled
```

```
(a .)        ;Illegal; an error is signaled
(a .. b)     ;Illegal; an error is signaled
(a . . b)    ;Illegal; an error is signaled
(a b c ...)  ;Illegal; an error is signaled
```

In all other cases, the token is construed to be the name of a symbol. If there are any package markers (colons) in the token, they divide the token into pieces used to control the lookup and creation of the symbol.

If there is a single package marker, and it occurs at the beginning of the token, then the token is interpreted as a keyword, that is, a symbol in the `keyword` package. The part of the token after the package marker must not have the syntax of a number.

If there is a single package marker not at the beginning or end of the token, then it divides the token into two parts. The first part specifies a package; the second part is the name of an external symbol available in that package. Neither of the two parts may have the syntax of a number.

If there are two adjacent package markers not at the beginning or end of the token, then they divide the token into two parts. The first part specifies a package; the second part is the name of a symbol within that package (possibly an internal symbol). Neither of the two parts may have the syntax of a number.

X3J13 voted in March 1988 to clarify that, in the situations described in the preceding three paragraphs, the restriction on the syntax of the parts should be strengthened: none of the parts may have the syntax of even a *potential* number. Tokens such as `:3600`, `:1/2`, and `editor:3.14159` were already ruled out; this clarification further declares that such tokens as `:2^3`, `compiler:1.7J`, and `Christmas:12/25/83` are also in error and therefore should not be used in portable programs. Implementations may differ in their treatment of such package-marked potential numbers.

If a symbol token contains no package markers, then the entire token is the name of the symbol. The symbol is looked up in the default package, which is the value of the variable `*package*`.

All other patterns of package markers, including the cases where there are more than two package markers or where a package marker appears at the end of the token, at present do not mean anything in Common Lisp (see chapter 11). It is therefore currently an error to use such patterns in a

Common Lisp program. The valid patterns for tokens may be summarized as follows:

nnnnn	a number
xxxxx	a symbol in the current package
`:`*xxxxx*	a symbol in the keyword package
ppppp`:`*xxxxx*	an external symbol in the *ppppp* package
ppppp`::`*xxxxx*	a (possibly internal) symbol in the *ppppp* package

where *nnnnn* has the syntax of a number, and *xxxxx* and *ppppp* do not have the syntax of a number.

In accordance with the X3J13 decision noted above , *xxxxx* and *ppppp* may not have the syntax of even a potential number.

[Variable] ***read-base***

The value of `*read-base*` controls the interpretation of tokens by `read` as being integers or ratios. Its value is the radix in which integers and ratios are to be read; the value may be any integer from `2` to `36` (inclusive) and is normally `10` (decimal radix). Its value affects only the reading of integers and ratios. In particular, floating-point numbers are always read in decimal radix. The value of `*read-base*` does not affect the radix for rational numbers whose radix is explicitly indicated by `#O`, `#X`, `#B`, or `#`*n*`R` syntax or by a trailing decimal point.

Care should be taken when setting `*read-base*` to a value larger than `10`, because tokens that would normally be interpreted as symbols may be interpreted as numbers instead. For example, with `*read-base*` set to `16` (hexadecimal radix), variables with names such as `a`, `b`, `f`, `bad`, and `face` will be treated by the reader as numbers (with decimal values 10, 11, 15, 2989, and 64206, respectively). The ability to alter the input radix is provided in Common Lisp primarily for the purpose of reading data files in special operatorats, rather than for the purpose of altering the default radix in which to read programs. The user is strongly encouraged to use `#O`, `#X`, `#B`, or `#`*n*`R` syntax when notating non-decimal constants in programs.

[Variable] ***read-suppress***

When the value of `*read-suppress*` is `nil`, the Lisp reader operates normally. When it is not `nil`, then most of the interesting operations of the

reader are suppressed; input characters are parsed, but much of what is read is not interpreted.

The primary purpose of `*read-suppress*` is to support the operation of the read-time conditional constructs `#+` and `#-` (see section 22.1.4). It is important for these constructs to be able to skip over the printed representation of a Lisp expression despite the possibility that the syntax of the skipped expression may not be entirely legal for the current implementation; this is because a primary application of `#+` and `#-` is to allow the same program to be shared among several Lisp implementations despite small incompatibilities of syntax.

A non-`nil` value of `*read-suppress*` has the following specific effects on the Common Lisp reader:

- All extended tokens are completely uninterpreted. It matters not whether the token looks like a number, much less like a valid number; the pattern of package markers also does not matter. An extended token is simply discarded and treated as if it were `nil`; that is, reading an extended token when `*read-suppress*` is non-`nil` simply returns `nil`. (One consequence of this is that the error concerning improper dotted-list syntax will not be signaled.)

- Any standard `#` macro-character construction that requires, permits, or disallows an infix numerical argument, such as `#`*n*`R`, will not enforce any constraint on the presence, absence, or value of such an argument.

- The `#\` construction always produces the value `nil`. It will not signal an error even if an unknown character name is seen.

- Each of the `#B`, `#O`, `#X`, and `#R` constructions always scans over a following token and produces the value `nil`. It will not signal an error even if the token does not have the syntax of a rational number.

- The `#*` construction always scans over a following token and produces the value `nil`. It will not signal an error even if the token does not consist solely of the characters `0` and `1`.

- The `#.` construction reads the following form (in suppressed mode, of course) but does not evaluate it. The form is discarded and `nil` is produced.

- Each of the `#A`, `#S`, and `#:` constructions reads the following form (in suppressed mode, of course) but does not interpret it in any way; it need not even be a list in the case of `#S`, or a symbol in the case of `#:`. The form is discarded and `nil` is produced.
- The `#=` construction is totally ignored. It does not read a following form. It produces no object, but is treated as whitespace.
- The `##` construction always produces `nil`.

Note that, no matter what the value of `*read-suppress*`, parentheses still continue to delimit (and construct) lists; the `#(` construction continues to delimit vectors; and comments, strings, and the quote and backquote constructions continue to be interpreted properly. Furthermore, such situations as `')`, `#<`, `#)`, and `#`⟨space⟩ continue to signal errors.

In some cases, it may be appropriate for a user-written macro-character definition to check the value of `*read-suppress*` and to avoid certain computations or side effects if its value is not `nil`.

[Variable] ***read-eval***

Default value of `*read-eval*` is `t`. If `*read-eval*` is false, the `#.` reader macro signals an error.

Printing is also affected. If `*read-eval*` is false and `*print-readably*` is true, any `print-object` method that would otherwise output a `#.` reader macro must either output something different or signal an error of type `print-not-readable`.

Binding `*read-eval*` to `nil` is useful when reading data that came from an untrusted source, such as a network or a user-supplied data file; it prevents the `#.` reader macro from being exploited as a "Trojan horse" to cause arbitrary forms to be evaluated.

22.1.3 Macro Characters

If the reader encounters a macro character, then the function associated with that macro character is invoked and may produce an object to be returned. This function may read following characters in the stream in whatever syntax

it likes (it may even call `read` recursively) and return the object represented by that syntax. Macro characters may or may not be recognized, of course, when read as part of other special syntaxes (such as for strings).

The reader is therefore organized into two parts: the basic dispatch loop, which also distinguishes symbols and numbers, and the collection of macro characters. Any character can be reprogrammed as a macro character; this is a means by which the reader can be extended. The macro characters normally defined are as follows:

`(` The left-parenthesis character initiates reading of a pair or list. The function `read` is called recursively to read successive objects until a right parenthesis is found to be next in the input stream. A list of the objects read is returned. Thus the input sequence

(a b c)

is read as a list of three objects (the symbols `a`, `b`, and `c`). The right parenthesis need not immediately follow the printed representation of the last object; whitespace characters and comments may precede it. This can be useful for putting one object on each line and making it easy to add new objects:

```
(defun traffic-light (color)
  (case color
    (green)
    (red (stop))
    (amber (accelerate))     ;Insert more colors after this line
    ))
```

It may be that *no* objects precede the right parenthesis, as in `()` or `( )`; this reads as a list of zero objects (the empty list).

If a token that is just a dot, not preceded by an escape character, is read after some object, then exactly one more object must follow the dot, possibly followed by whitespace, followed by the right parenthesis:

(a b c . d)

This means that the *cdr* of the last pair in the list is not `nil`, but rather the object whose representation followed the dot. The above example might have been the result of evaluating

(cons 'a (cons 'b (cons 'c 'd))) ⇒ (a b c . d)

Similarly, we have

(cons 'znets 'wolq-zorbitan) ⇒ (znets . wolq-zorbitan)

It is permissible for the object following the dot to be a list:

(a b c d . (e f . (g)))

is the same as

(a b c d e f g)

but a list following a dot is a non-standard form that `print` will never produce.

`)` The right-parenthesis character is part of various constructs (such as the syntax for lists) using the left-parenthesis character and is invalid except when used in such a construct.

' The single-quote (accent acute) character provides an abbreviation to make it easier to put constants in programs. The form `'foo` reads the same as `(quote foo)`: a list of the symbol `quote` and *foo*.

; Semicolon is used to write comments. The semicolon and all characters up to and including the next newline are ignored. Thus a comment can be put at the end of any line without affecting the reader. (A comment will terminate a token, but a newline would terminate the token anyway.)

There is no functional difference between using one semicolon and using more than one, but the conventions shown here are in common use.

```
;;;; COMMENT-EXAMPLE function.
;;; This function is useless except to demonstrate comments.
;;; (Actually, this example is much too cluttered with them.)

(defun comment-example (x y)      ;X is anything; Y is an a-list.
  (cond ((listp x) x)             ;If X is a list, use that.
        ;; X is now not a list. There are two other cases.
        ((symbolp x)
         ;; Look up a symbol in the a-list.
         (cdr (assoc x y)))       ;Remember, (cdr nil) is nil.
        ;; Do this when all else fails:
        (t (cons x                ;Add x to a default list.
                 '((lisp t)       ;LISP is okay.
                   (fortran nil)  ;FORTRAN is not.
                   (pl/i -500)    ;Note that you can put comments in
                   (ada .001)     ; "data" as well as in "programs".
                   ;; COBOL??
                   (teco -1.0e9))))))
```

In this example, comments may begin with one to four semicolons.

- Single-semicolon comments are all aligned to the same column at the right; usually each comment concerns only the code it is next to. Occasionally a comment is long enough to occupy two or three lines; in this case, it is conventional to indent the continued lines of the comment one space (after the semicolon).

- Double-semicolon comments are aligned to the level of indentation of the code. A space conventionally follows the two semicolons. Such comments usually describe the state of the program at that point or the code section that follows the comment.
- Triple-semicolon comments are aligned to the left margin. They usually document whole programs or large code blocks.
- Quadruple-semicolon comments usually indicate titles of whole programs or large code blocks.

" The double quote character begins the printed representation of a string. Successive characters are read from the input stream and accumulated until another double quote is encountered. An exception to this occurs if a *single escape* character is seen; the escape character is discarded, the next character is accumulated, and accumulation continues. When a matching double quote is seen, all the accumulated characters up to but not including the matching double quote are made into a simple string and returned.

‘ The backquote (accent grave) character makes it easier to write programs to construct complex data structures by using a template. *Notice of correction.* In the first edition, the backquote character ⟨‘⟩ appearing at the left margin above was inadvertently omitted. As an example, writing

```
‘(cond ((numberp ,x) ,@y) (t (print ,x) ,@y))
```

is roughly equivalent to writing

```
(list 'cond
      (cons (list 'numberp x) y)
      (list* 't (list 'print x) y))
```

The general idea is that the backquote is followed by a template, a picture of a data structure to be built. This template is copied, except that within the template commas can appear. Where a comma occurs,

the form following the comma is to be evaluated to produce an object to be inserted at that point. Assume `b` has the value `3`; then evaluating the form denoted by `‘(a b ,b ,(+ b 1) b)` produces the result `(a b 3 4 b)`.

If a comma is immediately followed by an at-sign (`@`), then the form following the at-sign is evaluated to produce a *list* of objects. These objects are then "spliced" into place in the template. For example, if `x` has the value `(a b c)`, then

‘(x ,x ,@x foo ,(cadr x) bar ,(cdr x) baz ,@(cdr x))
⇒ (x (a b c) a b c foo b bar (b c) baz b c)

The backquote syntax can be summarized formally as follows. For each of several situations in which backquote can be used, a possible interpretation of that situation as an equivalent form is given. Note that the form is equivalent only in the sense that when it is evaluated it will calculate the correct result. An implementation is quite free to interpret backquote in any way such that a backquoted form, when evaluated, will produce a result `equal` to that produced by the interpretation shown here.

- ‘*basic* is the same as ’*basic*, that is, `(quote` *basic*`)`, for any form *basic* that is not a list or a general vector.
- ‘,*form* is the same as *form*, for any *form*, provided that the representation of *form* does not begin with "`@`" or "`.`". (A similar caveat holds for all occurrences of a form after a comma.)
- ‘,@*form* is an error.
- ‘(*x1 x2 x3 ... xn . atom*`)` may be interpreted to mean

  (append [*x1*] [*x2*] [*x3*] ... [*xn*] (quote *atom*))

  where the brackets are used to indicate a transformation of an *xj* as follows:
  - [*form*] is interpreted as `(list ‘`*form*`)`, which contains a backquoted form that must then be further interpreted.

  - [`,`*form*] is interpreted as `(list` *form*`)`.
  - [`,@`*form*] is interpreted simply as *form*.

- `‘(`*x1 x2 x3 ... xn*`)` may be interpreted to mean the same as the backquoted form `‘(`*x1 x2 x3 ... xn* `. nil)`, thereby reducing it to the previous case.
- `‘(`*x1 x2 x3 ... xn* `. ,`*form*`)` may be interpreted to mean

  (append [*x1*] [*x2*] [*x3*] ... [*xn*] *form*)

  where the brackets indicate a transformation of an *xj* as described above.
- `‘(`*x1 x2 x3 ... xn* `. ,@`*form*`)` is an error.
- `‘#(`*x1 x2 x3 ... xn*`)` may be interpreted to mean

  (apply #'vector ‘(*x1 x2 x3 ... xn*))

No other uses of comma are permitted; in particular, it may not appear within the `#A` or `#S` syntax.

Anywhere "`,@`" may be used, the syntax "`,.`" may be used instead to indicate that it is permissible to destroy the list produced by the form following the "`,.`"; this may permit more efficient code, using `nconc` instead of `append`, for example.

If the backquote syntax is nested, the innermost backquoted form should be expanded first. This means that if several commas occur in a row, the leftmost one belongs to the innermost backquote.

Once again, it is emphasized that an implementation is free to interpret a backquoted form as any form that, when evaluated, will produce a result that is `equal` to the result implied by the above definition. In particular, no guarantees are made as to whether the constructed copy of the template will or will not share list structure with the template itself. As an example, the above definition implies that

‘((,a b) ,c ,@d)

will be interpreted as if it were

```
(append (list (append (list a) (list 'b) 'nil)) (list c) d 'nil)
```

but it could also be legitimately interpreted to mean any of the following.

```
(append (list (append (list a) (list 'b))) (list c) d)
(append (list (append (list a) '(b))) (list c) d)
(append (list (cons a '(b))) (list c) d)
(list* (cons a '(b)) c d)
(list* (cons a (list 'b)) c d)
(list* (cons a '(b)) c (copy-list d))
```

(There is no good reason why `copy-list` should be performed, but it is not prohibited.)

Some users complain that backquote syntax is difficult to read, especially when it is nested. I agree that it can get complicated, but in some situations (such as writing macros that expand into definitions for other macros) such complexity is to be expected, and the alternative is much worse.

After I gained some experience in writing nested backquote forms, I found that I was not stopping to analyze the various patterns of nested backquotes and interleaved commas and quotes; instead, I was recognizing standard idioms wholesale, in the same manner that I recognize `cadar` as the primitive for "extract the lambda-list from the form `((lambda ...) ...)`" without stopping to analyze it into "`car` of `cdr` of `car`." For example, `,x` within a doubly-nested backquote form means "the value of `x` available during the second evaluation will appear here once the form has been twice evaluated," whereas `,',x` means "the value of `x` available during the first evaluation will appear here once the form has been twice evaluated" and `,,x` means "the value of the value of `x` will appear here."

See appendix **??** for a systematic set of examples of the use of nested backquotes.

`,` The comma character is part of the backquote syntax and is invalid if used other than inside the body of a backquote construction as described above.

`#` This is a *dispatching* macro character. It reads an optional digit string and then one more character, and uses that character to select a function to run as a macro-character function.

The `#` character also happens to be a non-terminating macro character. This is completely independent of the fact that it is a dispatching macro character; it is a coincidence that the only standard dispatching macro character in Common Lisp is also the only standard non-terminating macro character.

See the next section for predefined `#` macro-character constructions.

22.1.4 Standard Dispatching Macro Character Syntax

The standard syntax includes forms introduced by the `#` character. These take the general form of a `#`, a second character that identifies the syntax, and following arguments in some form. If the second character is a letter, then case is not important; `#O` and `#o` are considered to be equivalent, for example.

Certain `#` forms allow an unsigned decimal number to appear between the `#` and the second character; some other forms even require it. Those forms that do not explicitly permit such a number to appear forbid it.

The currently defined `#` constructs are described below and summarized in table 22.8; more are likely to be added in the future. However, the constructs `#!`, `#?`, `#[`, `#]`, `#{`, and `#}` are explicitly reserved for the user and will never be defined by the Common Lisp standard.

`#\` `#\`*x* reads in as a character object that represents the character *x*. Also, `#\`*name* reads in as the character object whose name is *name*. Note that the backslash `\` allows this construct to be parsed easily by EMACS-like editors.

In the single-character case, the character *x* must be followed by a non-constituent character, lest a *name* appear to follow the `#\`. A good model of what happens is that after `#\` is read, the reader backs up over the `\` and then reads an extended token, treating the initial `\` as an escape character (whether it really is or not in the current readtable).

Uppercase and lowercase letters are distinguished after `#\`; `#\A` and `#\a` denote different character objects. Any character works after `#\`, even those that are normally special to `read`, such as parentheses. Non-printing characters may be used after `#\`, although for them names are generally preferred.

`#\`*name* reads in as a character object whose name is *name* (actually, whose name is `(string-upcase` *name*`)`; therefore the syntax is case-insensitive). The *name* should have the syntax of a symbol. The following names are standard across all implementations:

`newline`	The character that represents the division between lines
`space`	The space or blank character

Таблица 22.7: Standard # Macro Character Syntax

`#!`	*undefined* *	`#`⟨backspace⟩	*signals error*
`#"`	*undefined*	`#`⟨tab⟩	*signals error*
`##`	*reference to* `#=` *label*	`#`⟨newline⟩	*signals error*
`#$`	*undefined*	`#`⟨linefeed⟩	*signals error*
`#%`	*undefined*	`#`⟨page⟩	*signals error*
`#&`	*undefined*	`#`⟨return⟩	*signals error*
`#'`	`function` *abbreviation*	`#`⟨space⟩	*signals error*
`#(`	*simple vector*	`#+`	*read-time conditional*
`#)`	*signals error*	`#-`	*read-time conditional*
`#*`	*bit-vector*	`#.`	*read-time evaluation*
`#,`	*load-time evaluation*	`#/`	*undefined*
`#0`	*used for infix arguments*	`#A`, `#a`	*array*
`#1`	*used for infix arguments*	`#B`, `#b`	*binary rational*
`#2`	*used for infix arguments*	`#C`, `#c`	*complex number*
`#3`	*used for infix arguments*	`#D`, `#d`	*undefined*
`#4`	*used for infix arguments*	`#E`, `#e`	*undefined*
`#5`	*used for infix arguments*	`#F`, `#f`	*undefined*
`#6`	*used for infix arguments*	`#G`, `#g`	*undefined*
`#7`	*used for infix arguments*	`#H`, `#h`	*undefined*
`#8`	*used for infix arguments*	`#I`, `#i`	*undefined*
`#9`	*used for infix arguments*	`#J`, `#j`	*undefined*
`#:`	*uninterned symbol*	`#K`, `#k`	*undefined*
`#;`	*undefined*	`#L`, `#l`	*undefined*
`#<`	*signals error*	`#M`, `#m`	*undefined*
`#=`	*label following object*	`#N`, `#n`	*undefined*
`#>`	*undefined*	`#O`, `#o`	*octal rational*
`#?`	*undefined* *	`#P`, `#p`	*pathname*
`#@`	*undefined*	`#Q`, `#q`	*undefined*
`#[`	*undefined* *	`#R`, `#r`	*radix-n rational*
`#\`	*character object*	`#S`, `#s`	*structure*
`#]`	*undefined* *	`#T`, `#t`	*undefined*
`#^`	*undefined*	`#U`, `#u`	*undefined*
`#_`	*undefined*	`#V`, `#v`	*undefined*
`#‘`	*undefined*	`#W`, `#w`	*undefined*
`#{`	*undefined* *	`#X`, `#x`	*hexadecimal rational*
`#|`	*balanced comment*	`#Y`, `#y`	*undefined*
`#}`	*undefined* *	`#Z`, `#z`	*undefined*
`#~`	*undefined*	`#`⟨rubout⟩	*undefined*

The combinations marked by an asterisk are explicitly reserved to the user and will never be defined by Common Lisp.

X3J13 voted in June 1989 to specify `#P` and `#p` (*undefined* in the first edition).

Таблица 22.8: Стандартный синтаксис для макросимвола #

#!*неопределен* *	#⟨backspace⟩	*сигнализирует ошибку*
#"*неопределен*	#⟨tab⟩	*сигнализирует ошибку*
##*ссылка на #= метку*	#⟨newline⟩	*сигнализирует ошибку*
#$*неопределен*	#⟨linefeed⟩	*сигнализирует ошибку*
#%*неопределен*	#⟨page⟩	*сигнализирует ошибку*
#&*неопределен*	#⟨return⟩	*сигнализирует ошибку*
#'*аббревиатуры для* `function`	#⟨space⟩	*сигнализирует ошибку*
#(*простой вектор*	#+	*условное вычисление во*
#)*сигнализирует ошибку*	#-	*условное вычисление во*
#**битовый вектор*	#.	*вычисление во время чт*
#,*вычисление во время загрузки*	#/	*неопределен*
#0*используется для инфиксных аргументов*	#A, #a	*массив*
#1*используется для инфиксных аргументов*	#B, #b	*двоичное число*
#2*используется для инфиксных аргументов*	#C, #c	*комплексное число*
#3*используется для инфиксных аргументов*	#D, #d	*неопределен*
#4*используется для инфиксных аргументов*	#E, #e	*неопределен*
#5*используется для инфиксных аргументов*	#F, #f	*неопределен*
#6*используется для инфиксных аргументов*	#G, #g	*неопределен*
#7*используется для инфиксных аргументов*	#H, #h	*неопределен*
#8*используется для инфиксных аргументов*	#I, #i	*неопределен*
#9*используется для инфиксных аргументов*	#J, #j	*неопределен*
#:*uninterned symbol*	#K, #k	*неопределен*
#;*неопределен*	#L, #l	*неопределен*
#<*сигнализирует ошибку*	#M, #m	*неопределен*
#=*метка следующего объекта*	#N, #n	*неопределен*
#>*неопределен*	#O, #o	*восьмеричное число*
#?*неопределен* *	#P, #p	*имя файла*
#@*неопределен*	#Q, #q	*неопределен*
#[*неопределен* *	#R, #r	*число с основанием*
#*символьный объект*	#S, #s	*структура*
#]*неопределен* *	#T, #t	*неопределен*
#^*неопределен*	#U, #u	*неопределен*
#_*неопределен*	#V, #v	*неопределен*
#‘*неопределен*	#W, #w	*неопределен*
#{*неопределен* *	#X, #x	*шестнадцатиричное чи*
#|*многострочный комментарий*	#Y, #y	*неопределен*
#}*неопределен* *	#Z, #z	*неопределен*
#~*неопределен*	#⟨rubout⟩	*неопределен*

Комбинации обозначенные звёздочкой зарезервированы для использования пользователем. Такие комбинации никогда не будут использованы стандартом Common Lisp'a.

The following names are semi-standard; if an implementation supports them, they should be used for the described characters and no others.

`rubout`	The rubout or delete character.
`page`	The form-feed or page-separator character
`tab`	The tabulate character
`backspace`	The backspace character
`return`	The carriage return character
`linefeed`	The line-feed character

In some implementations, one or more of these characters might be a synonym for a standard character; the `#\Linefeed` character might be the same as `#\Newline`, for example.

When the Lisp printer types out the name of a special character, it uses the same table as the `#\` reader; therefore any character name you see typed out is acceptable as input (in that implementation). Standard names are always preferred over non-standard names for printing.

The following convention is used in implementations that support non-zero bits attributes for character objects. If a name after `#\` is longer than one character and has a hyphen in it, then it may be split into the two parts preceding and following the first hyphen; the first part (actually, `string-upcase` of the first part) may then be interpreted as the name or initial of a bit, and the second part as the name of the character (which may in turn contain a hyphen and be subject to further splitting). For example:

#\Control-Space	#\Control-Meta-Tab
#\C-M-Return	#\H-S-M-C-Rubout

If the character name consists of a single character, then that character is used. Another `\` may be necessary to quote the character.

#\Control-%	#\Control-Meta-\"
#\Control-\a	#\Meta->

If an unsigned decimal integer appears between the `#` and `\`, it is interpreted as a font number, to become the font attribute of the character object (see `char-font`).

X3J13 voted in March 1989 to replace the notion of bits and font attributes with that of implementation-defined attributes. Presumably this eliminates the portable use of this syntax for font information, although the vote did not address this question directly.

`#'` `#'`*foo* is an abbreviation for `(function` *foo*`)`. *foo* may be the printed representation of any Lisp object. This abbreviation may be remembered by analogy with the `'` macro character, since the `function` and `quote` special operators are similar in form.

`#(` A series of representations of objects enclosed by `#(` and `)` is read as a simple vector of those objects. This is analogous to the notation for lists.

If an unsigned decimal integer appears between the `#` and `(`, it specifies explicitly the length of the vector. In that case, it is an error if too many objects are specified before the closing `)`, and if too few are specified, the last object (it is an error if there are none in this case) is used to fill all remaining elements of the vector. For example,

#(a b c c c c) #6(a b c c c c) #6(a b c) #6(a b c c)

all mean the same thing: a vector of length 6 with elements `a`, `b`, and four instances of `c`. The notation `#()` denotes an empty vector, as does `#0()` (which is legitimate because it is not the case that too few elements are specified).

`#*` A series of binary digits (`0` and `1`) preceded by `#*` is read as a simple bit-vector containing those bits, the leftmost bit in the series being bit 0 of the bit-vector.

If an unsigned decimal integer appears between the `#` and `*`, it specifies explicitly the length of the vector. In that case, it is an error if too many bits are specified, and if too few are specified the last one (it is an error if there are none in this case) is used to fill all remaining elements of the bit-vector. For example,

#*101111 #6*101111 #6*101 #6*1011

all mean the same thing: a vector of length 6 with elements `1`, `0`, `1`, `1`, `1`, and `1`. The notation `#*` denotes an empty bit-vector, as does `#0*` (which is legitimate because it is not the case that too few elements are specified). Compare this to `#B`, used for expressing integers in binary notation.

`#:` `#:`*foo* requires *foo* to have the syntax of an unqualified symbol name (no embedded colons). It denotes an *uninterned* symbol whose name is *foo*. Every time this syntax is encountered, a different uninterned symbol is created. If it is necessary to refer to the same uninterned symbol more than once in the same expression, the `#=` syntax may be useful.

`#.` `#.`*foo* is read as the object resulting from the evaluation of the Lisp object represented by *foo*, which may be the printed representation of any Lisp object. The evaluation is done during the `read` process, when the `#.` construct is encountered.

X3J13 voted in June 1989 to add a new reader control variable, `*read-eval*`. If it is true, the `#.` reader macro behaves as described above; if it is false, the `#.` reader macro signals an error.

The `#.` syntax therefore performs a read-time evaluation of *foo*. By contrast, `#,` (see below) performs a load-time evaluation.

Both `#.` and `#,` allow you to include, in an expression being read, an object that does not have a convenient printed representation; instead of writing a representation for the object, you write an expression that will *compute* the object.

`#B` `#b`*rational* reads *rational* in binary (radix 2). For example, `#B1101` ≡ `13`, and `#b101/11` ≡ `5/3`. Compare this to `#*`, used for expressing bit-vectors in binary notation.

`#O` `#o`*rational* reads *rational* in octal (radix 8). For example, `#o37/15` ≡ `31/13`, and `#o777` ≡ `511`.

`#X` `#x`*rational* reads *rational* in hexadecimal (radix 16). The digits above `9` are the letters `A` through `F` (the lowercase letters `a` through `f` are also acceptable). For example, `#xF00` ≡ `3840`.

`#`*n*`R` `#`*radix*`r`*rational* reads *rational* in radix *radix*. *radix* must consist of only digits, and it is read in decimal; its value must be between 2 and 36 (inclusive).

For example, `#3r102` is another way of writing `11`, and `#11R32` is another way of writing `35`. For radices larger than 10, letters of the alphabet are used in order for the digits after `9`.

`#nA` The syntax `#nA`*object* constructs an n-dimensional array, using *object* as the value of the `:initial-contents` argument to `make-array`.

The value of n makes a difference: `#2A((0 1 5) (foo 2 (hot dog)))`, for example, represents a 2-by-3 matrix:

```
0      1      5
foo    2      (hot dog)
```

In contrast, `#1A((0 1 5) (foo 2 (hot dog)))` represents a length-2 array whose elements are lists:

```
(0 1 5)    (foo 2 (hot dog))
```

Furthermore, `#0A((0 1 5) (foo 2 (hot dog)))` represents a zero-dimensional array whose sole element is a list:

```
((0 1 5) (foo 2 (hot dog)))
```

Similarly, `#0Afoo` (or, more readably, `#0A foo`) represents a zero-dimensional array whose sole element is the symbol `foo`. The expression `#1Afoo` would not be legal because `foo` is not a sequence.

`#S` The syntax `#s(`*name slot1 value1 slot2 value2 ...*`)` denotes a structure. This is legal only if *name* is the name of a structure already defined by `defstruct` and if the structure has a standard constructor macro, which it normally will. Let *cm* stand for the name of this constructor macro; then this syntax is equivalent to

#.(*cm keyword1 'value1 keyword2 'value2 ...*)

where each *keywordj* is the result of computing

(intern (string *slotj*) 'keyword)

(This computation is made so that one need not write a colon in front of every slot name.) The net effect is that the constructor macro is called with the specified slots having the specified values (note that one does not write quote marks in the `#S` syntax). Whatever object the constructor macro returns is returned by the `#S` syntax.

`#P` X3J13 voted in June 1989 to define the reader syntax `#p"..."` to be equivalent to `#.(parse-namestring "...")`. Presumably this was meant to be taken descriptively and not literally. I would think, for example, that the committee did not wish to quibble over the package in which the name `parse-namestring` was to be read. Similarly, I would presume that the `#p` syntax operates normally rather than signaling an error when `*read-eval*` is false. I interpret the intent of the vote to be that `#p` reads a following form, which should be a string, that is then converted to a pathname as if by application of the standard function `parse-namestring`.

`#`*n*`=` The syntax `#`*n*`=`*object* reads as whatever Lisp object has *object* as its printed representation. However, that object is labelled by *n*, a required unsigned decimal integer, for possible reference by the syntax `#`*n*`#` (below). The scope of the label is the expression being read by the outermost call to `read`. Within this expression the same label may not appear twice.

`#`*n*`#` The syntax `#`*n*`#`, where *n* is a required unsigned decimal integer, serves as a reference to some object labelled by `#`*n*`=`; that is, `#`*n*`#` represents a pointer to the same identical (`eq`) object labelled by `#`*n*`=`. This permits notation of structures with shared or circular substructure. For example, a structure created in the variable `y` by this code:

```
(setq x (list 'p 'q))
(setq y (list (list 'a 'b) x 'foo x))
(rplacd (last y) (cdr y))
```

could be represented in this way:

((a b) . #1=(#2=(p q) foo #2# . #1#))

Without this notation, but with `*print-length*` set to `10`, the structure would print in this way:

((a b) (p q) foo (p q) (p q) foo (p q) (p q) foo (p q) ...)

A reference `#n#` may occur only after a label `#n=`; forward references are not permitted. In addition, the reference may not appear as the labelled object itself (that is, one may not write `#n= #n#`), because the object labelled by `#n=` is not well defined in this case.

`#+` The `#+` syntax provides a read-time conditionalization facility; the syntax is

#+*feature form*

If *feature* is "true," then this syntax represents a Lisp object whose printed representation is *form*. If *feature* is "false," then this syntax is effectively whitespace; it is as if it did not appear.

The *feature* should be the printed representation of a symbol or list. If *feature* is a symbol, then it is true if and only if it is a member of the list that is the value of the global variable `*features*`.

Otherwise, *feature* should be a Boolean expression composed of `and`, `or`, and `not` operators on (recursive) *feature* expressions.

For example, suppose that in implementation A the features `spice` and `perq` are true, and in implementation B the feature `lispm` is true. Then the expressions on the left below are read the same as those on the right in implementation A:

```
(cons #+spice "Spice" #+lispm "Lispm" x)      (cons "Spice" x)
(setq a '(1 2 #+perq 43 #+(not perq) 27))     (setq a '(1 2 43))
(let ((a 3) #+(or spice lispm) (b 3))         (let ((a 3) (b 3))
  (foo a))                                      (foo a))
(cons a #+perq #-perq b c)                    (cons a c)
```

In implementation B, however, they are read in this way:

```
(cons #+spice "Spice" #+lispm "Lispm" x)      (cons "Lispm" x)
(setq a '(1 2 #+perq 43 #+(not perq) 27))     (setq a '(1 2 27))
(let ((a 3) #+(or spice lispm) (b 3))         (let ((a 3) (b 3))
  (foo a))                                      (foo a))
(cons a #+perq #-perq b c)                    (cons a c)
```

The `#+` construction must be used judiciously if unreadable code is not to result. The user should make a careful choice between read-time conditionalization and run-time conditionalization.

The `#+` syntax operates by first reading the *feature* specification and then skipping over the *form* if the *feature* is "false." This skipping of a form is a bit tricky because of the possibility of user-defined macro characters and side effects caused by the `#.` construction. It is accomplished by binding the variable `*read-suppress*` to a non-`nil` value and then calling the `read` function. See the description of `*read-suppress*` for the details of this operation.

X3J13 voted in March 1988 to specify that the `keyword` package is the default package during the reading of a feature specification. Thus `#+spice` means the same thing as `#+:spice`, and `#+(or spice lispm)` means the same thing as `#+(or :spice :lispm)`. Symbols in other packages may be used as feature names, but one must use an explicit package prefix to cite one after `#+`.

`#-` `#-`*feature form* is equivalent to `#+(not` *feature*`)` *form*.

`#|` `#|...|#` is treated as a comment by the reader, just as everything from a semicolon to the next newline is treated as a comment. Anything may appear in the comment, except that it must be balanced with respect to other occurrences of `#|` and `|#`. Except for this nesting rule, the comment may contain any characters whatsoever.

The main purpose of this construct is to allow "commenting out" of blocks of code or data. The balancing rule allows such blocks to contain pieces already so commented out. In this respect the `#|...|#` syntax of Common Lisp differs from the `/*...*/` comment syntax used by PL/I and C.

`#<` This is not legal reader syntax. It is conventionally used in the printed representation of objects that cannot be read back in. Attempting to read a `#<` will cause an error. (More precisely, it *is* legal syntax, but the macro-character function for `#<` signals an error.)

The usual convention for printing unreadable data objects is to print some identifying information (the internal machine address of the object, if nothing else) preceded by `#<` and followed by `>`.

X3J13 voted in June 1989 to add `print-unreadable-object`, a macro that prints an object using `#<...>` syntax and also takes care of checking the variable `*print-readably*`.

#⟨space⟩, #⟨tab⟩, #⟨newline⟩, #⟨page⟩, #⟨return⟩ A # followed by a whitespace character is not legal reader syntax. This prevents abbreviated forms produced via `*print-level*` cutoff from reading in again, as a safeguard against losing information. (More precisely, this *is* legal syntax, but the macro-character function for it signals an error.)

`#)` This is not legal reader syntax. This prevents abbreviated forms produced via `*print-level*` cutoff from reading in again, as a safeguard against losing information. (More precisely, this *is* legal syntax, but the macro-character function for it signals an error.)

22.1.5 The Readtable

Previous sections describe the standard syntax accepted by the `read` function. This section discusses the advanced topic of altering the standard syntax either to provide extended syntax for Lisp objects or to aid the writing of other parsers.

There is a data structure called the *readtable* that is used to control the reader. It contains information about the syntax of each character equivalent to that in table 22.2. It is set up exactly as in table 22.2 to give the standard Common Lisp meanings to all the characters, but the user can change the meanings of characters to alter and customize the syntax of characters. It is also possible to have several readtables describing different syntaxes and to switch from one to another by binding the variable `*readtable*`.

[Variable] ***readtable***

The value of `*readtable*` is the current readtable. The initial value of this is a readtable set up for standard Common Lisp syntax. You can bind this variable to temporarily change the readtable being used.

To program the reader for a different syntax, a set of functions are provided for manipulating readtables. Normally, you should begin with a copy of the standard Common Lisp readtable and then customize the individual characters within that copy.

[Function] **copy-readtable** **&optional** *from-readtable to-readtable*

A copy is made of *from-readtable*, which defaults to the current readtable (the value of the global variable `*readtable*`). If *from-readtable* is `nil`, then a copy of a standard Common Lisp readtable is made. For example,

```
(setq *readtable* (copy-readtable nil))
```

will restore the input syntax to standard Common Lisp syntax, even if the original readtable has been clobbered (assuming it is not so badly clobbered that you cannot type in the above expression!). On the other hand,

```
(setq *readtable* (copy-readtable))
```

will merely replace the current readtable with a copy of itself.

If *to-readtable* is unsupplied or `nil`, a fresh copy is made. Otherwise, *to-readtable* must be a readtable, which is destructively copied into.

[Function] **readtablep** *object*

`readtablep` is true if its argument is a readtable, and otherwise is false.

(readtablep x) ≡ (typep x 'readtable)

[Function] **set-syntax-from-char** *to-char from-char* **&optional** *to-readtable from-readtable*

This makes the syntax of *to-char* in *to-readtable* be the same as the syntax of *from-char* in *from-readtable*. The *to-readtable* defaults to the current readtable (the value of the global variable `*readtable*`), and *from-readtable* defaults to `nil`, meaning to use the syntaxes from the standard Lisp readtable. X3J13 voted in January 1989 to clarify that the *to-char* and *from-char* must each be a character.

Only attributes as shown in table 22.2 are copied; moreover, if a *macro character* is copied, the macro definition function is copied also. However, attributes as shown in table 22.6 are not copied; they are "hard-wired" into the extended-token parser. For example, if the definition of `S` is copied to `*`, then `*` will become a *constituent* that is *alphabetic* but cannot be used as an exponent indicator for short-format floating-point number syntax.

It works to copy a macro definition from a character such as " to another character; the standard definition for " looks for another character that is the same as the character that invoked it. It doesn't work to copy the definition of (to {, for example; it can be done, but it lets one write lists in the form {a b c), not {a b c}, because the definition always looks for a closing parenthesis, not a closing brace. See the function read-delimited-list, which is useful in this connection.

The set-syntax-from-char function returns t.

[Function] **set-macro-character** *char function* **&optional** *non-terminating-p readtable*
[Function] **get-macro-character** *char* **&optional** *readtable*

set-macro-character causes *char* to be a macro character that when seen by read causes *function* to be called. If *non-terminating-p* is not nil (it defaults to nil), then it will be a non-terminating macro character: it may be embedded within extended tokens. set-macro-character returns t.

get-macro-character returns the function associated with *char* and, as a second value, returns the *non-terminating-p* flag; it returns nil if *char* does not have macro-character syntax. In each case, *readtable* defaults to the current readtable.

If nil is explicitly passed as the second argument to get-macro-character, then the standard readtable is used. This is consistent with the behavior of copy-readtable.

The *function* is called with two arguments, *stream* and *char*. The *stream* is the input stream, and *char* is the macro character itself. In the simplest case, *function* may return a Lisp object. This object is taken to be that whose printed representation was the macro character and any following characters read by the *function*. As an example, a plausible definition of the standard single quote character is:

```
(defun single-quote-reader (stream char)
  (declare (ignore char))
  (list 'quote (read stream t nil t)))

(set-macro-character #\' #'single-quote-reader)
```

(Note that `t` is specified for the *recursive-p* argument to `read`; see section 22.2.1.) The function reads an object following the single-quote and returns a list of the symbol `quote` and that object. The *char* argument is ignored.

The function may choose instead to return *zero* values (for example, by using `(values)` as the return expression). In this case, the macro character and whatever it may have read contribute nothing to the object being read. As an example, here is a plausible definition for the standard semicolon (comment) character:

```
(defun semicolon-reader (stream char)
  (declare (ignore char))
  ;; First swallow the rest of the current input line.
  ;; End-of-file is acceptable for terminating the comment.
  (do () ((char= (read-char stream nil #\Newline t) #\Newline)))
  ;; Return zero values.
  (values))

(set-macro-character #\; #'semicolon-reader)
```

(Note that `t` is specified for the *recursive-p* argument to `read-char`; see section 22.2.1.)

The *function* should not have any side effects other than on the *stream*. Because of backtracking and restarting of the `read` operation, front ends (such as editors and rubout handlers) to the reader may cause *function* to be called repeatedly during the reading of a single expression in which the macro character only appears once.

Here is an example of a more elaborate set of read-macro characters that I used in the implementation of the original simulator for Connection Machine Lisp [44, 57], a parallel dialect of Common Lisp. This simulator was used to gain experience with the language before freezing its design for full-scale implementation on a Connection Machine computer system. This example illustrates the typical manner in which a language designer can embed a new language within the syntactic and semantic framework of Lisp, saving the effort of designing an implementation from scratch.

Connection Machine Lisp introduces a new data type called a *xapping*, which is simply an unordered set of ordered pairs of Lisp objects. The first element of each pair is called the *index* and the second element the *value*. We say that the xapping maps each index to its corresponding value. No two pairs of the same xapping may have the same (that is, `eql`) index. Xappings may be finite or infinite sets of pairs; only certain kinds of infinite xappings are required, and special representations are used for them.

A finite xapping is notated by writing the pairs between braces, separated by whitespace. A pair is notated by writing the index and the value, separated by a right arrow (or an exclamation point if the host Common Lisp has no right-arrow character).

Примечание: The original language design used the right arrow; the exclamation point was chosen to replace it on ASCII-only terminals because it is one of the six characters `[ ] { } ! ?` reserved by Common Lisp to the user.

While preparing the TEX manuscript for this book I made a mistake in font selection and discovered that by an absolutely incredible coincidence the right arrow has the same numerical code (octal 41) within TEX fonts as the ASCII exclamation point. The result was that although the manuscript called for right arrows, exclamation points came out in the printed copy. Imagine my astonishment!

Here is an example of a xapping that maps three symbols to strings:

{moe⇒"Oh, a wise guy, eh?" larry⇒"Hey, what's the idea?"
 curly⇒"Nyuk, nyuk, nyuk!"}

For convenience there are certain abbreviated notations. If the index and value for a pair are the same object x, then instead of having to write "x⇒ x" (or, worse yet, "`#43=`*x*⇒`#43#`") we may write simply x for the pair. If all pairs of a xapping are of this form, we call the xapping a *xet*. For example, the notation

{baseball chess cricket curling bocce 43-man-squamish}

is entirely equivalent in meaning to

{baseball⇒baseball curling⇒curling cricket⇒cricket
 chess⇒chess bocce⇒bocce 43-man-squamish⇒43-man-squamish}

namely a xet of symbols naming six sports.

Another useful abbreviation covers the situation where the n pairs of a finite xapping are integers, collectively covering a range from zero to $n-1$. This kind of xapping is called a *xector* and may be notated by writing the values between brackets in ascending order of their indices. Thus

[tinker evers chance]

is merely an abbreviation for

{tinker⇒0 evers⇒1 chance⇒2}

There are two kinds of infinite xapping: constant and universal. A constant xapping `{⇒`*z*`}` maps every object to the same value *z*. The universal xapping `{⇒}` maps every object to itself and is therefore the xet of all Lisp objects, sometimes called simply the universe. Both kinds of infinite xet may be modified by explicitly writing exceptions. One kind of exception is simply a pair, which specifies the value for a particular index; the other kind of exception is simply *k*⇒ indicating that the xapping does *not* have a pair with index *k* after all. Thus the notation

{sky⇒blue grass⇒green idea⇒ glass⇒ ⇒red}

indicates a xapping that maps `sky` to `blue`, `grass` to `green`, and every other object except `idea` and `glass` to `red`. Note well that the presence or absence of whitespace on either side of an arrow is crucial to the correct interpretation of the notation.

Here is the representation of a xapping as a structure:

```
(defstruct
  (xapping (:print-function print-xapping)
           (:constructor xap
             (domain range &optional
              (default ':unknown defaultp)
              (infinite (and defaultp :constant))
              (exceptions '()))))
  domain
  range
  default
  (infinite nil :type (member nil :constant :universal)
  exceptions)
```

The explicit pairs are represented as two parallel lists, one of indexes (`domain`) and one of values (`range`). The `default` slot is the default value, relevant only if the `infinite` slot is `:constant`. The `exceptions` slot is a list of indices for which there are no values. (See the end of section 22.3.3 for the definition of `print-xapping`.)

Here, then, is the code for reading xectors in bracket notation:

```
(defun open-bracket-macro-char (stream macro-char)
  (declare (ignore macro-char))
  (let ((range (read-delimited-list #\] stream t)))
    (xap (iota-list (length range)) range)))

(set-macro-character #\[ #'open-bracket-macro-char)
(set-macro-character #\] (get-macro-character #\) ))

(defun iota-list (n)     ;Return list of integers from 0 to n − 1
  (do ((j (- n 1) (- j 1))
       (z '() (cons j z)))
      ((< j 0) z)))
```

The code for reading xappings in the more general brace notation, with all the possibilities for xets (or individual xet pairs), infinite xappings, and exceptions, is a bit more complicated; it is shown in table 22.9. That code is used in conjunction with the initializations

```
(set-macro-character #\{ #'open-brace-macro-char)
(set-macro-character #\} (get-macro-character #\) ))
```

[Function] **make-dispatch-macro-character** *char* **&optional** *non-terminating-p readtable*

This causes the character *char* to be a dispatching macro character in *readtable* (which defaults to the current readtable). If *non-terminating-p* is not `nil` (it defaults to `nil`), then it will be a non-terminating macro character: it may be embedded within extended tokens. `make-dispatch-macro-character` returns `t`.

Таблица 22.9: Macro Character Definition for Xapping Syntax

```
(defun open-brace-macro-char (s macro-char)
  (declare (ignore macro-char))
  (do ((ch (peek-char t s t nil t) (peek-char t s t nil t))
       (domain '())  (range '())  (exceptions '()))
      ((char= ch #\})
       (read-char s t nil t)
       (construct-xapping (reverse domain) (reverse range)))
    (cond ((char= ch #\⇒)
           (read-char s t nil t)
           (let ((nextch (peek-char nil s t nil t)))
             (cond ((char= nextch #\})
                    (read-char s t nil t)
                    (return (xap (reverse domain)
                                 (reverse range)
                                 nil :universal exceptions)))
                   (t (let ((item (read s t nil t)))
                        (cond ((char= (peek-char t s t nil t) #\})
                               (read-char s t nil t)
                               (return (xap (reverse domain)
                                            (reverse range)
                                            item :constant
                                            exceptions)))
                              (t (reader-error s
                                   "Default ⇒ item must be last")))))))))
          (t (let ((item (read-preserving-whitespace s t nil t))
                   (nextch (peek-char nil s t nil t)))
               (cond ((char= nextch #\⇒)
                      (read-char s t nil t)
                      (cond ((member (peek-char nil s t nil t)
                                     '(#\Space #\Tab #\Newline))
                             (push item exceptions))
                            (t (push item domain)
                               (push (read s t nil t) range))))
                     ((char= nch #\})
                      (read-char s t nil t)
                      (push item domain)
                      (push item range)
                      (return (xap (reverse domain) (reverse range))))
                     (t (push item domain)
                        (push item range)))))))))
```

Initially every character in the dispatch table has a character-macro function that signals an error. Use `set-dispatch-macro-character` to define entries in the dispatch table. X3J13 voted in January 1989 to clarify that *char* must be a character.

[Function] **set-dispatch-macro-character** *disp-char sub-char function* **&optional** *readtable*
[Function] **get-dispatch-macro-character** *disp-char sub-char* **&optional** *readtable*

`set-dispatch-macro-character` causes *function* to be called when the *disp-char* followed by *sub-char* is read. The *readtable* defaults to the current readtable. The arguments and return values for *function* are the same as for normal macro characters except that *function* gets *sub-char*, not *disp-char*, as its second argument and also receives a third argument that is the non-negative integer whose decimal representation appeared between *disp-char* and *sub-char*, or `nil` if no decimal integer appeared there.

The *sub-char* may not be one of the ten decimal digits; they are always reserved for specifying an infix integer argument. Moreover, if *sub-char* is a lowercase character (see `lower-case-p`), its uppercase equivalent is used instead. (This is how the rule is enforced that the case of a dispatch sub-character doesn't matter.)

`set-dispatch-macro-character` returns `t`.

`get-dispatch-macro-character` returns the macro-character function for *sub-char* under *disp-char*, or `nil` if there is no function associated with *sub-char*.

If the *sub-char* is one of the ten decimal digits `0 1 2 3 4 5 6 7 8 9`, `get-dispatch-macro-character` always returns `nil`. If *sub-char* is a lowercase character, its uppercase equivalent is used instead.

X3J13 voted in January 1989 to specify that if `nil` is explicitly passed as the second argument to `get-dispatch-macro-character`, then the standard readtable is used. This is consistent with the behavior of `copy-readtable`.

For either function, an error is signaled if the specified *disp-char* is not in fact a dispatch character in the specified readtable. It is necessary to use `make-dispatch-macro-character` to set up the dispatch character before specifying its sub-characters.

As an example, suppose one would like `#$`*`foo`* to be read as if it were `(dollars` *`foo`*`)`. One might say:

```
(defun |#$-reader| (stream subchar arg)
  (declare (ignore subchar arg))
  (list 'dollars (read stream t nil t)))

(set-dispatch-macro-character #\# #\$ #'|#$-reader|)
```

[Function] **readtable-case** *readtable*

X3J13 voted in June 1989 to introduce the function `readtable-case` to control the reader's interpretation of case. It provides access to a slot in a readtable, and may be used with `setf` to alter the state of that slot. The possible values for the slot are `:upcase`, `:downcase`, `:preserve`, and `:invert`; the `readtable-case` for the standard readtable is `:upcase`. Note that `copy-readtable` is required to copy the `readtable-case` slot along with all other readtable information.

Once the reader has accumulated a token as described in section 22.1.1, if the token is a symbol, "replaceable" characters (unescaped uppercase or lowercase constituent characters) may be modified under the control of the `readtable-case` of the current readtable:

- For `:upcase`, replaceable characters are converted to uppercase. (This was the behavior specified by the first edition.)
- For `:downcase`, replaceable characters are converted to lowercase.
- For `:preserve`, the cases of all characters remain unchanged.
- For `:invert`, if all of the replaceable letters in the extended token are of the same case, they are all converted to the opposite case; otherwise the cases of all characters in that token remain unchanged.

As an illustration, consider the following code.

```
(let ((*readtable* (copy-readtable nil)))
  (format t "READTABLE-CASE  Input    Symbol-name~
          ~%————————   ————————~
          ~%")
  (dolist (readtable-case '(:upcase :downcase :preserve :invert))
    (setf (readtable-case *readtable*) readtable-case)
    (dolist (input '("ZEBRA" "Zebra" "zebra"))
```

```
(format t ":~A~16T~A~24T~A~%"
        (string-upcase readtable-case)
        input
        (symbol-name (read-from-string input)))))))
```

The output from this test code should be

```
READTABLE-CASE  Input   Symbol-name
————————————————————————————————
:UPCASE         ZEBRA   ZEBRA
:UPCASE         Zebra   ZEBRA
:UPCASE         zebra   ZEBRA
:DOWNCASE       ZEBRA   zebra
:DOWNCASE       Zebra   zebra
:DOWNCASE       zebra   zebra
:PRESERVE       ZEBRA   ZEBRA
:PRESERVE       Zebra   Zebra
:PRESERVE       zebra   zebra
:INVERT         ZEBRA   zebra
:INVERT         Zebra   Zebra
:INVERT         zebra   ZEBRA
```

The `readtable-case` of the current readtable also affects the printing of symbols (see `*print-case*` and `*print-escape*`).

22.1.6 What the Print Function Produces

The Common Lisp printer is controlled by a number of special variables. These are referred to in the following discussion and are fully documented at the end of this section.

How an expression is printed depends on its data type, as described in the following paragraphs.

Integers If appropriate, a radix specifier may be printed; see the variable `*print-radix*`. If an integer is negative, a minus sign is printed and

then the absolute value of the integer is printed. Integers are printed in the radix specified by the variable `*print-base*` in the usual positional notation, most significant digit first. The number zero is represented by the single digit `0` and never has a sign. A decimal point may then be printed, depending on the value of `*print-radix*`.

Ratios If appropriate, a radix specifier may be printed; see the variable `*print-radix*`. If the ratio is negative, a minus sign is printed. Then the absolute value of the numerator is printed, as for an integer; then a `/`; then the denominator. The numerator and denominator are both printed in the radix specified by the variable `*print-base*`; they are obtained as if by the `numerator` and `denominator` functions, and so ratios are always printed in reduced form (lowest terms).

Floating-point numbers If the sign of the number (as determined by the function `float-sign`) is negative, then a minus sign is printed. Then the magnitude is printed in one of two ways. If the magnitude of the floating-point number is either zero or between 10^{-3} (inclusive) and 10^{7} (exclusive), it may be printed as the integer part of the number, then a decimal point, followed by the fractional part of the number; there is always at least one digit on each side of the decimal point. If the format of the number does not match that specified by the variable `*read-default-float-format*`, then the exponent marker for that format and the digit `0` are also printed. For example, the base of the natural logarithms as a short-format floating-point number might be printed as `2.71828S0`.

For non-zero magnitudes outside of the range 10^{-3} to 10^{7}, a floating-point number will be printed in "computerized scientific notation." The representation of the number is scaled to be between 1 (inclusive) and 10 (exclusive) and then printed, with one digit before the decimal point and at least one digit after the decimal point. Next the exponent marker for the format is printed, except that if the format of the number matches that specified by the variable `*read-default-float-format*`, then the exponent marker `E` is used. Finally, the power of 10 by which the fraction must be multiplied to equal the original number is printed as a decimal integer. For example, Avogadro's number as a short-format floating-point number might be printed as `6.02S23`.

Complex numbers A complex number is printed as `#C`, an open parenthesis, the printed representation of its real part, a space, the printed representation of its imaginary part, and finally a close parenthesis.

Characters When `*print-escape*` is `nil`, a character prints as itself; it is sent directly to the output stream. When `*print-escape*` is not `nil`, then `#\` syntax is used. For example, the printed representation of the character `#\A` with control and meta bits on would be `#\CONTROL-META-A`, and that of `#\a` with control and meta bits on would be `#\CONTROL-META-\a`. X3J13 voted in June 1989 to specify that if `*print-readably*` is not `nil` then every object must be printed in a readable form, regardless of other printer control variables. For characters, the simplest approach is always to use `#\` syntax when `*print-readably*` is not `nil`, regardless of the value of `*print-escape*`.

Symbols When `*print-escape*` is `nil`, only the characters of the print name of the symbol are output (but the case in which to print any uppercase characters in the print name is controlled by the variable `*print-case*`).

X3J13 voted in June 1989 to specify that the new `readtable-case` slot of the current readtable also controls the case in which letters (whether uppercase or lowercase) in the print name of a symbol are output, no matter what the value of `*print-escape*`.

The remaining paragraphs describing the printing of symbols cover the situation when `*print-escape*` is not `nil`.

X3J13 voted in June 1989 to specify that if `*print-readably*` is not `nil` then every object must be printed in a readable form, regardless of other printer control variables. For symbols, the simplest approach is to print them, when `*print-readably*` is not `nil`, as if `*print-escape*` were not `nil`, regardless of the actual value of `*print-escape*`.

Backslashes \ and vertical bars | are included as required. In particular, backslash or vertical-bar syntax is used when the name of the symbol would be otherwise treated by the reader as a potential number (see section 22.1.2). In making this decision, it is assumed that the value of `*print-base*` being used for printing would be used as the value of `*read-base*` used for reading; the value of `*read-base*` at the time

of printing is irrelevant. For example, if the value of `*print-base*` were `16` when printing the symbol `face`, it would have to be printed as `\FACE` or `\Face` or `|FACE|`, because the token `face` would be read as a hexadecimal number (decimal value 64206) if `*read-base*` were `16`.

The case in which to print any uppercase characters in the print name is controlled by the variable `*print-case*`. X3J13 voted in June 1989 to clarify the interaction of `*print-case*` with `*print-escape*`; see `*print-case*`. As a special case [no pun intended], `nil` may sometimes be printed as `()` instead, when `*print-escape*` and `*print-pretty*` are both not `nil`.

Package prefixes may be printed (using colon syntax) if necessary. The rules for package qualifiers are as follows. When the symbol is printed, if it is in the keyword package, then it is printed with a preceding colon; otherwise, if it is accessible in the current package, it is printed without any qualification; otherwise, it is printed with qualification. See chapter 11.

A symbol that is uninterned (has no home package) is printed preceded by `#:` if the variables `*print-gensym*` and `*print-escape*` are both non-`nil`; if either is `nil`, then the symbol is printed without a prefix, as if it were in the current package. X3J13 voted in June 1989 to specify that if `*print-readably*` is not `nil` then every object must be printed in a readable form, regardless of other printer control variables. For uninterned symbols, the simplest approach is to print them, when `*print-readably*` is not `nil`, as if `*print-escape*` and `*print-gensym*` were not `nil`, regardless of their actual values.

Заметка для реализации: Because the `#:` syntax does not intern the following symbol, it is necessary to use circular-list syntax if `*print-circle*` is not `nil` and the same uninterned symbol appears several times in an expression to be printed. For example, the result of

```
(let ((x (make-symbol "FOO"))) (list x x))
```

would be printed as

```
(#:foo #:foo)
```

if `*print-circle*` were `nil`, but as

```
(#1=#:foo #1#)
```

if `*print-circle*` were not `nil`.

The case in which symbols are to be printed is controlled by the variable `*print-case*`. It is also controlled by `*print-escape*` and the `readtable-case` slot of the current readtable (the value of `*readtable*`).

Strings The characters of the string are output in order. If `*print-escape*` is not `nil`, a double quote is output before and after, and all double quotes and single escape characters are preceded by backslash. The printing of strings is not affected by `*print-array*`. If the string has a fill pointer, then only those characters below the fill pointer are printed.

X3J13 voted in June 1989 to specify that if `*print-readably*` is not `nil` then every object must be printed in a readable form, regardless of other printer control variables. For strings, the simplest approach is to print them, when `*print-readably*` is not `nil`, as if `*print-escape*` were not `nil`, regardless of the actual value of `*print-escape*`.

Conses Wherever possible, list notation is preferred over dot notation. Therefore the following algorithm is used:

1. Print an open parenthesis, `(`.
2. Print the *car* of the cons.
3. If the *cdr* is a cons, make it the current cons, print a space, and go to step 2.
4. If the *cdr* is not null, print a space, a dot, a space, and the *cdr*.
5. Print a close parenthesis, `)`.

This form of printing is clearer than showing each individual cons cell. Although the two expressions below are equivalent, and the reader will accept either one and produce the same data structure, the printer will always print such a data structure in the second form.

(a . (b . ((c . (d . `nil`)) . (e . `nil`))))

(a b (c d) e)

The printing of conses is affected by the variables `*print-level*` and `*print-length*`.

X3J13 voted in June 1989 to specify that if `*print-readably*` is not `nil` then every object must be printed in a readable form, regardless of other printer control variables. For conses, the simplest approach is to print them, when `*print-readably*` is not `nil`, as if `*print-level*` and `*print-length*` were `nil`, regardless of their actual values.

Bit-vectors A bit-vector is printed as `#*` followed by the bits of the bit-vector in order. If `*print-array*` is `nil`, however, then the bit-vector is printed in a format (using `#<`) that is concise but not readable. If the bit-vector has a fill pointer, then only those bits below the fill pointer are printed. X3J13 voted in June 1989 to specify that if `*print-readably*` is not `nil` then every object must be printed in a readable form, regardless of other printer control variables. For bit-vectors, the simplest approach is to print them, when `*print-readably*` is not `nil`, as if `*print-array*` were not `nil`, regardless of the actual value of `*print-array*`.

Vectors Any vector other than a string or bit-vector is printed using general-vector syntax; this means that information about specialized vector representations will be lost. The printed representation of a zero-length vector is `#()`. The printed representation of a non-zero-length vector begins with `#(`. Following that, the first element of the vector is printed. If there are any other elements, they are printed in turn, with a space printed before each additional element. A close parenthesis after the last element terminates the printed representation of the vector. The printing of vectors is affected by the variables `*print-level*` and `*print-length*`. If the vector has a fill pointer, then only those elements below the fill pointer are printed.

If `*print-array*` is `nil`, however, then the vector is not printed as described above, but in a format (using `#<`) that is concise but not readable. X3J13 voted in June 1989 to specify that if `*print-readably*`

is not `nil` then every object must be printed in a readable form, regardless of other printer control variables. For vectors, the simplest approach is to print them, when `*print-readably*` is not `nil`, as if `*print-level*` and `*print-length*` were `nil` and `*print-array*` were not `nil`, regardless of their actual values.

Arrays Normally any array other than a vector is printed using `#`*n*`A` format. Let n be the rank of the array. Then `#` is printed, then n as a decimal integer, then `A`, then n open parentheses. Next the elements are scanned in row-major order. Imagine the array indices being enumerated in odometer fashion, recalling that the dimensions are numbered from 0 to $n-1$. Every time the index for dimension j is incremented, the following actions are taken:

1. If $j < n-1$, then print a close parenthesis.
2. If incrementing the index for dimension j caused it to equal dimension j, reset that index to zero and increment dimension $j-1$ (thereby performing these three steps recursively), unless $j = 0$, in which case simply terminate the entire algorithm. If incrementing the index for dimension j did not cause it to equal dimension j, then print a space.
3. If $j < n-1$, then print an open parenthesis.

This causes the contents to be printed in a format suitable for use as the `:initial-contents` argument to `make-array`. The lists effectively printed by this procedure are subject to truncation by `*print-level*` and `*print-length*`.

If the array is of a specialized type, containing bits or string-characters, then the innermost lists generated by the algorithm given above may instead be printed using bit-vector or string syntax, provided that these innermost lists would not be subject to truncation by `*print-length*`. For example, a 3-by-2-by-4 array of string-characters that would ordinarily be printed as

```
#3A(((#\s #\t #\o #\p) (#\s #\p #\o #\t))
   ((#\p #\o #\s #\t) (#\p #\o #\t #\s))
   ((#\t #\o #\p #\s) (#\o #\p #\t #\s)))
```

may instead be printed more concisely as

```
#3A(("stop" "spot") ("post" "pots") ("tops" "opts"))
```

If `*print-array*` is `nil`, then the array is printed in a format (using `#<`) that is concise but not readable. X3J13 voted in June 1989 to specify that if `*print-readably*` is not `nil` then every object must be printed in a readable form, regardless of other printer control variables. For arrays, the simplest approach is to print them, when `*print-readably*` is not `nil`, as if `*print-level*` and `*print-length*` were `nil` and `*print-array*` were not `nil`, regardless of their actual values.

Random-states Common Lisp does not specify a specific syntax for printing objects of type `random-state`. However, every implementation must arrange to print a random-state object in such a way that, within the same implementation of Common Lisp, the function `read` can construct from the printed representation a copy of the random-state object as if the copy had been made by `make-random-state`.

Pathnames If `*print-escape*` is true, a pathname should be printed by `write` as `#P"..."` where `"..."` is the namestring representation of the pathname. If `*print-escape*` is false, `write` prints a pathname by printing its namestring (presumably without escape characters or surrounding double quotes).

If `*print-readably*` is not `nil` then every object must be printed in a readable form, regardless of other printer control variables. For pathnames, the simplest approach is to print them, when `*print-readably*` is not `nil`, as if `*print-escape*` were `nil`, regardless of its actual value.

Structures defined by `defstruct` are printed under the control of the user-specified `:print-function` option to `defstruct`. If the user does not provide a printing function explicitly, then a default printing function is supplied that prints the structure using `#S` syntax (see section 22.1.4).

If `*print-readably*` is not `nil` then every object must be printed in a readable form, regardless of the values of other printer control variables;

if this is not possible, then an error of type `print-not-readable` must be signaled to avoid printing an unreadable syntax such as `#<...>`.

Macro `print-unreadable-object` prints an object using `#<...>` syntax and also takes care of checking the variable `*print-readably*`.

When debugging or when frequently dealing with large or deep objects at top level, the user may wish to restrict the printer from printing large amounts of information. The variables `*print-level*` and `*print-length*` allow the user to control how deep the printer will print and how many elements at a given level the printer will print. Thus the user can see enough of the object to identify it without having to wade through the entire expression.

[Variable] ***print-readably***

The default value of `*print-readably*` is `nil`. If `*print-readably*` is true, then printing any object must either produce a printed representation that the reader will accept or signal an error. If printing is successful, the reader will, on reading the printed representation, produce an object that is "similar as a constant" (see section 24.1.4) to the object that was printed.

If `*print-readably*` is true and printing a readable printed representation is not possible, the printer signals an error of type `print-not-readable` rather than using an unreadable syntax such as `#<`. The printed representation produced when `*print-readably*` is true might or might not be the same as the printed representation produced when `*print-readably*` is false.

If `*print-readably*` is true and another printer control variable (such as `*print-length*`, `*print-level*`, `*print-escape*`, `*print-gensym*`, `*print-array*`, or an implementation-defined printer control variable) would cause the preceding requirements to be violated, that other printer control variable is ignored.

The printing of interned symbols is not affected by `*print-readably*`.

Note that the "similar as a constant" rule for readable printing implies that `#A` or `#(` syntax cannot be used for arrays of element-type other than `t`. An implementation will have to use another syntax or signal a `print-not-readable` error. A `print-not-readable` error will not be signaled for strings or bit-vectors.

All methods for `print-object` must obey `*print-readably*`. This rule applies to both user-defined methods and implementation-defined methods.

The reader control variable `*read-eval*` also affects printing. If `*read-eval*` is false and `*print-readably*` is true, any `print-object` method that would otherwise output a `#.` reader macro must either output something different or signal an error of type `print-not-readable`.

Readable printing of structures and objects of type `standard-object` is controlled by their `print-object` methods, not by their `make-load-form` methods. "Similarity as a constant" for these objects is application-dependent and hence is defined to be whatever these methods do.

`*print-readably*` allows errors involving data with no readable printed representation to be detected when writing the file rather than later on when the file is read.

`*print-readably*` is more rigorous than `*print-escape*`; output printed with escapes must be merely generally recognizable by humans, with a good chance of being recognizable by computers, whereas output printed readably must be reliably recognizable by computers.

[Variable] ***print-escape***

When this flag is `nil`, then escape characters are not output when an expression is printed. In particular, a symbol is printed by simply printing the characters of its print name. The function `princ` effectively binds `*print-escape*` to `nil`.

When this flag is not `nil`, then an attempt is made to print an expression in such a way that it can be read again to produce an `equal` structure. The function `prin1` effectively binds `*print-escape*` to `t`. The initial value of this variable is `t`.

[Variable] ***print-pretty***

When this flag is `nil`, then only a small amount of whitespace is output when printing an expression.

When this flag is not `nil`, then the printer will endeavor to insert extra whitespace where appropriate to make the expression more readable. A few other simple changes may be made, such as printing `'foo` instead of `(quote foo)`.

The initial value of `*print-pretty*` is implementation-dependent.

X3J13 voted in January 1989 to adopt a facility for user-controlled pretty printing in Common Lisp (see chapter 26).

[Variable] ***print-circle***

When this flag is `nil` (the default), then the printing process proceeds by recursive descent; an attempt to print a circular structure may lead to looping behavior and failure to terminate.

If `*print-circle*` is true, the printer is required to detect not only cycles but shared substructure, indicating both through the use of `#n=` and `#n#` syntax. As an example, under the specification of the first edition

```
(print '(#1=(a #1#) #1#))
```

might legitimately print `(#1=(A #1#) #1#)` or `(#1=(A #1#) #2=(A #2#))`; the vote specifies that the first form is required.

User-defined printing functions for the `defstruct :print-function` option, as well as user-defined methods for the CLOS generic function `print-object`, may print objects to the supplied stream using `write`, `print1`, `princ`, `format`, or `print-object` and expect circularities to be detected and printed using `#n#` syntax (when `*print-circle*` is non-`nil`, of course).

It seems to me that the same ought to apply to abbreviation as controlled by `*print-level*` and `*print-length*`, but that was not addressed by this vote.

[Variable] ***print-base***

The value of `*print-base*` determines in what radix the printer will print rationals. This may be any integer from `2` to `36`, inclusive; the default value is `10` (decimal radix). For radices above `10`, letters of the alphabet are used to represent digits above `9`.

[Variable] ***print-radix***

If the variable `*print-radix*` is non-`nil`, the printer will print a radix specifier to indicate the radix in which it is printing a rational number. To prevent confusion of the letter `O` with the digit `0`, and of the letter `B` with the digit `8`, the radix specifier is always printed using lowercase letters. For example, if the current base is twenty-four (decimal), the decimal integer twenty-three would print as `#24rN`. If `*print-base*` is `2`, `8`, or `16`, then the radix specifier used is `#b`, `#o`, or `#x`. For integers, base ten is indicated by a

trailing decimal point instead of a leading radix specifier; for ratios, however, `#10r` is used. The default value of `*print-radix*` is `nil`.

[Variable] ***print-case***

The `read` function normally converts lowercase characters appearing in symbols to corresponding uppercase characters, so that internally print names normally contain only uppercase characters. However, users may prefer to see output using lowercase letters or letters of mixed case. This variable controls the case (upper, lower, or mixed) in which to print any uppercase characters in the names of symbols when vertical-bar syntax is not used. The value of `*print-case*` should be one of the keywords `:upcase`, `:downcase`, or `:capitalize`; the initial value is `:upcase`.

Lowercase characters in the internal print name are always printed in lowercase, and are preceded by a single escape character or enclosed by multiple escape characters. Uppercase characters in the internal print name are printed in uppercase, in lowercase, or in mixed case so as to capitalize words, according to the value of `*print-case*`. The convention for what constitutes a "word" is the same as for the function `string-capitalize`.

X3J13 voted in June 1989 to clarify the interaction of `*print-case*` with `*print-escape*`. When `*print-escape*` is `nil`, `*print-case*` determines the case in which to print all uppercase characters in the print name of the symbol. When `*print-escape*` is not `nil`, the implementation has some freedom as to which characters will be printed so as to appear in an "escape context" (after an escape character, typically `\`, or between multiple escape characters, typically `|`); `*print-case*` determines the case in which to print all uppercase characters that will not appear in an escape context. For example, when the value of `*print-case*` is `:upcase`, an implementation might choose to print the symbol whose print name is `"(S)HE"` as `\(S\)HE` or as `|(S)HE|`, among other possibilities. When the value of `*print-case*` is `:downcase`, the corresponding output should be `\(s\)he` or `|(S)HE|`, respectively.

Consider the following test code. (For the sake of this example assume that `readtable-case` is `:upcase` in the current readtable; this is discussed further below.)

```
(let ((tabwidth 11))
  (dolist (sym '(|x| |FoObAr| |fOo|))
    (let ((tabstop -1))
```

```
    (format t "~&")
    (dolist (escape '(t nil))
      (dolist (case '(:upcase :downcase :capitalize))
        (format t "~VT" (* (incf tabstop) tabwidth))
        (write sym :escape escape :case case)))))
  (format t " %"))
```

An implementation that leans heavily on multiple-escape characters (vertical bars) might produce the following output:

```
|x|       |x|       |x|       x         x         x
|FoObAr|  |FoObAr|  |FoObAr|  FoObAr    foobar    Foobar
|fOo|     |fOo|     |fOo|     fOo       foo       foo
```

An implementation that leans heavily on single-escape characters (backslashes) might produce the following output:

```
\x        \x        \x        x         x         x
F\oO\bA\r f\oo\ba\r F\oo\ba\r FoObAr    foobar    Foobar
\fO\o     \fo\o     \fo\o     fOo       foo       foo
```

These examples are not exhaustive; output using both kinds of escape characters (for example, `|FoO|\bA\r`) is permissible (though ugly).

X3J13 voted in June 1989 to add a new `readtable-case` slot to readtables to control automatic case conversion during the reading of symbols. The value of `readtable-case` in the current readtable also affects the printing of unescaped letters (letters appearing in an escape context are always printed in their own case).

- If `readtable-case` is `:upcase`, unescaped uppercase letters are printed in the case specified by `*print-case*` and unescaped lowercase letters are printed in their own case. (If `*print-escape*` is non-`nil`, all lowercase letters will necessarily be escaped.)

- If `readtable-case` is `:downcase`, unescaped lowercase letters are printed in the case specified by `*print-case*` and unescaped uppercase letters are printed in their own case. (If `*print-escape*` is non-`nil`, all uppercase letters will necessarily be escaped.)

- If `readtable-case` is `:preserve`, all unescaped letters are printed in their own case, regardless of the value of `*print-case*`. There is no need to escape any letters, even if `*print-escape*` is non-`nil`, though the X3J13 vote did not prohibit escaping letters in this situation.
- If `readtable-case` is `:invert`, and if all unescaped letters are of the same case, then the case of all the unescaped letters is inverted; but if the unescaped letters are not all of the same case then each is printed in its own case. (Thus `:invert` does not always invert the case; the inversion is conditional.) There is no need to escape any letters, even if `*print-escape*` is non-`nil`, though the X3J13 vote did not prohibit escaping letters in this situation.

Consider the following code.

```
;;; Generate a table illustrating READTABLE-CASE and *PRINT-CASE*.

(let ((*readtable* (copy-readtable nil))
      (*print-case* *print-case*))
  (format t "READTABLE-CASE *PRINT-CASE*  Symbol-name  Output~
          ~%-------------    -------------~
          ~%")
  (dolist (readtable-case '(:upcase :downcase :preserve :invert))
    (setf (readtable-case *readtable*) readtable-case)
    (dolist (print-case '(:upcase :downcase :capitalize))
      (dolist (sym '(|ZEBRA| |Zebra| |zebra|))
        (setq *print-case* print-case)
        (format t ":~A~15T:~A~29T~A~42T~A~%"
                (string-upcase readtable-case)
                (string-upcase print-case)
                (symbol-name sym)
                (prin1-to-string sym)))))))
```

Note that the call to `prin1-to-string` (the last argument in the call to `format` that is within the nested loops) effectively uses a non-`nil` value for `*print-escape*`.

Assuming an implementation that uses vertical bars around a symbol name if any characters need escaping, the output from this test code should be

READTABLE-CASE	*PRINT-CASE*	Symbol-name	Output
:UPCASE	:UPCASE	ZEBRA	ZEBRA
:UPCASE	:UPCASE	Zebra	|Zebra|
:UPCASE	:UPCASE	zebra	|zebra|
:UPCASE	:DOWNCASE	ZEBRA	zebra
:UPCASE	:DOWNCASE	Zebra	|Zebra|
:UPCASE	:DOWNCASE	zebra	|zebra|
:UPCASE	:CAPITALIZE	ZEBRA	Zebra
:UPCASE	:CAPITALIZE	Zebra	|Zebra|
:UPCASE	:CAPITALIZE	zebra	|zebra|
:DOWNCASE	:UPCASE	ZEBRA	|ZEBRA|
:DOWNCASE	:UPCASE	Zebra	|Zebra|
:DOWNCASE	:UPCASE	zebra	ZEBRA
:DOWNCASE	:DOWNCASE	ZEBRA	|ZEBRA|
:DOWNCASE	:DOWNCASE	Zebra	|Zebra|
:DOWNCASE	:DOWNCASE	zebra	zebra
:DOWNCASE	:CAPITALIZE	ZEBRA	|ZEBRA|
:DOWNCASE	:CAPITALIZE	Zebra	|Zebra|
:DOWNCASE	:CAPITALIZE	zebra	Zebra
:PRESERVE	:UPCASE	ZEBRA	ZEBRA
:PRESERVE	:UPCASE	Zebra	Zebra
:PRESERVE	:UPCASE	zebra	zebra
:PRESERVE	:DOWNCASE	ZEBRA	ZEBRA
:PRESERVE	:DOWNCASE	Zebra	Zebra
:PRESERVE	:DOWNCASE	zebra	zebra
:PRESERVE	:CAPITALIZE	ZEBRA	ZEBRA
:PRESERVE	:CAPITALIZE	Zebra	Zebra
:PRESERVE	:CAPITALIZE	zebra	zebra
:INVERT	:UPCASE	ZEBRA	zebra
:INVERT	:UPCASE	Zebra	Zebra
:INVERT	:UPCASE	zebra	ZEBRA
:INVERT	:DOWNCASE	ZEBRA	zebra
:INVERT	:DOWNCASE	Zebra	Zebra
:INVERT	:DOWNCASE	zebra	ZEBRA
:INVERT	:CAPITALIZE	ZEBRA	zebra
:INVERT	:CAPITALIZE	Zebra	Zebra
:INVERT	:CAPITALIZE	zebra	ZEBRA

This illustrates all combinations for `readtable-case` and `*print-case*`.

Таблица 22.10: Examples of Print Level and Print Length Abbreviation

v	*n*	Output
0	1	#
1	1	(if ...)
1	2	(if # ...)
1	3	(if # # ...)
1	4	(if # # #)
2	1	(if ...)
2	2	(if (member x ...) ...)
2	3	(if (member x y) (+ # 3) ...)
3	2	(if (member x ...) ...)
3	3	(if (member x y) (+ (car x) 3) ...)
3	4	(if (member x y) (+ (car x) 3) '(foo . #(a b c d ...)))
3	5	(if (member x y) (+ (car x) 3) '(foo . #(a b c d "Baz")))

[Variable] ***print-gensym***

The `*print-gensym*` variable controls whether the prefix `#:` is printed before symbols that have no home package. The prefix is printed if the variable is not `nil`. The initial value of `*print-gensym*` is `t`.

[Variable] ***print-level***
[Variable] ***print-length***

The `*print-level*` variable controls how many levels deep a nested data object will print. If `*print-level*` is `nil` (the initial value), then no control is exercised. Otherwise, the value should be an integer, indicating the maximum level to be printed. An object to be printed is at level `0`; its components (as of a list or vector) are at level `1`; and so on. If an object to be recursively printed has components and is at a level equal to or greater than the value of `*print-level*`, then the object is printed as simply `#`.

The `*print-length*` variable controls how many elements at a given level are printed. A value of `nil` (the initial value) indicates that there be no limit to the number of components printed. Otherwise, the value of `*print-length*` should be an integer. Should the number of elements of a data object exceed the value `*print-length*`, the printer will print

three dots, ..., in place of those elements beyond the number specified by `*print-length*`. (In the case of a dotted list, if the list contains exactly as many elements as the value of `*print-length*`, and in addition has the non-null atom terminating it, that terminating atom is printed rather than the three dots.)

`*print-level*` and `*print-length*` affect the printing not only of lists but also of vectors, arrays, and any other object printed with a list-like syntax. They do not affect the printing of symbols, strings, and bit-vectors.

The Lisp reader will normally signal an error when reading an expression that has been abbreviated because of level or length limits. This signal is given because the `#` dispatch character normally signals an error when followed by whitespace or `)`, and because `...` is defined to be an illegal token, as are all tokens consisting entirely of periods (other than the single dot used in dot notation).

As an example, table 22.10 shows the ways the object

```
(if (member x y) (+ (car x) 3) '(foo . #(a b c d "Baz")))
```

would be printed for various values of `*print-level*` (in the column labeled v) and `*print-length*` (in the column labeled n).

[Variable] ***print-array***

If `*print-array*` is `nil`, then the contents of arrays other than strings are never printed. Instead, arrays are printed in a concise form (using `#<`) that gives enough information for the user to be able to identify the array but does not include the entire array contents. If `*print-array*` is not `nil`, non-string arrays are printed using `#(`, `#*`, or `#n A` syntax. *Notice of correction.* In the first edition, the preceding paragraph mentioned the nonexistent variable `print-array` instead of `*print-array*`. The initial value of `*print-array*` is implementation-dependent.

[Макрос] **with-standard-io-syntax** {declaration}* {form}*

Within the dynamic extent of the body, all reader/printer control variables, including any implementation-defined ones not specified by Common Lisp, are bound to values that produce standard read/print behavior. Table 22.12 shows the values to which standard Common Lisp variables are bound.

The values returned by `with-standard-io-syntax` are the values of the last body *form*, or `nil` if there are no body forms.

The intent is that a pair of executions, as shown in the following example, should provide reasonable reliable communication of data from one Lisp process to another:

```
;;; Write DATA to a file.
(with-open-file (file pathname :direction :output)
  (with-standard-io-syntax
    (print data file)))

;;; ... Later, in another Lisp:
(with-open-file (file pathname :direction :input)
  (with-standard-io-syntax
    (setq data (read file))))
```

Using `with-standard-io-syntax` to bind all the variables, instead of using `let` and explicit bindings, ensures that nothing is overlooked and avoids problems with implementation-defined reader/printer control variables. If the user wishes to use a non-standard value for some variable, such as `*package*` or `*read-eval*`, it can be bound by `let` inside the body of `with-standard-io-syntax`. For example:

```
;;; Write DATA to a file. Forbid use of #. syntax.
(with-open-file (file pathname :direction :output)
  (let ((*read-eval* nil))
    (with-standard-io-syntax
      (print data file))))

;;; Read DATA from a file. Forbid use of #. syntax.
(with-open-file (file pathname :direction :input)
  (let ((*read-eval* nil))
    (with-standard-io-syntax
      (setq data (read file)))))
```

Similarly, a user who dislikes the arbitrary choice of values for `*print-circle*` and `*print-pretty*` can bind these variables to other values inside the body.

The X3J13 vote left it unclear whether `with-standard-io-syntax` permits declarations to appear before the body of the macro call. I believe that was the intent, and this is reflected in the syntax shown above; but this is only my interpretation.

22.2 Input Functions

The input functions are divided into two groups: those that operate on streams of characters and those that operate on streams of binary data.

22.2.1 Input from Character Streams

Many character input functions take optional arguments called *input-stream*, *eof-error-p*, and *eof-value*. The *input-stream* argument is the stream from which to obtain input; if unsupplied or `nil` it defaults to the value of the special variable `*standard-input*`. One may also specify `t` as a stream, meaning the value of the special variable `*terminal-io*`.

The *eof-error-p* argument controls what happens if input is from a file (or any other input source that has a definite end) and the end of the file is reached. If *eof-error-p* is true (the default), an error will be signaled at end of file. If it is false, then no error is signaled, and instead the function returns *eof-value*.

An *eof-value* argument may be any Lisp datum whatsoever.

Functions such as `read` that read the representation of an object rather than a single character will always signal an error, regardless of *eof-error-p*, if the file ends in the middle of an object representation. For example, if a file does not contain enough right parentheses to balance the left parentheses in it, `read` will complain. If a file ends in a symbol or a number immediately followed by end-of-file, `read` will read the symbol or number successfully and when called again will see the end-of-file and only then act according to *eof-error-p*. Similarly, the function `read-line` will successfully read the last line of a file even if that line is terminated by end-of-file rather than the newline

character. If a file contains ignorable text at the end, such as blank lines and comments, `read` will not consider it to end in the middle of an object. Thus an *eof-error-p* argument controls what happens when the file ends *between* objects.

Many input functions also take an argument called *recursive-p*. If specified and not `nil`, this argument specifies that this call is not a "top-level" call to `read` but an imbedded call, typically from the function for a macro character. It is important to distinguish such recursive calls for three reasons.

First, a top-level call establishes the context within which the `#n=` and `#n#` syntax is scoped. Consider, for example, the expression

```
(cons '#3=(p q r) '(x y . #3#))
```

If the single-quote macro character were defined in this way:

```
(set-macro-character #\'
                     #'(lambda (stream char)
                         (declare (ignore char))
                         (list 'quote (read stream))))
```

then the expression could not be read properly, because there would be no way to know when `read` is called recursively by the first occurrence of `'` that the label `#3=` would be referred to later in the containing expression. There would be no way to know because `read` could not determine that it was called by a macro-character function rather than from "top level." The correct way to define the single quote macro character uses the *recursive-p* argument:

```
(set-macro-character #\'
                     #'(lambda (stream char)
                         (declare (ignore char))
                         (list 'quote (read stream t nil t))))
```

Second, a recursive call does not alter whether the reading process is to preserve whitespace or not (as determined by whether the top-level call was to `read` or `read-preserving-whitespace`). Suppose again that the single quote had the first, incorrect, macro-character definition shown above. Then a call to `read-preserving-whitespace` that read the expression `'foo` would fail to preserve the space character following the symbol `foo` because the single-quote macro-character function calls `read`, not `read-preserving-whitespace`, to read the following expression (in this case `foo`). The correct definition, which passes the value `t` for the *recursive-p* argument to `read`, allows the top-level call to determine whether whitespace is preserved.

Third, when end-of-file is encountered and the *eof-error-p* argument is not `nil`, the kind of error that is signaled may depend on the value of *recursive-p*. If *recursive-p* is not `nil`, then the end-of-file is deemed to have occurred within the middle of a printed representation; if *recursive-p* is `nil`, then the end-of-file may be deemed to have occurred between objects rather than within the middle of one.

[Function] **read &optional** *input-stream eof-error-p eof-value recursive-p*

`read` reads in the printed representation of a Lisp object from *input-stream*, builds a corresponding Lisp object, and returns the object.

Note that when the variable `*read-suppress*` is not `nil`, then `read` reads in a printed representation as best it can, but most of the work of interpreting the representation is avoided (the intent being that the result is to be discarded anyway). For example, all extended tokens produce the result `nil` regardless of their syntax.

[Variable] ***read-default-float-format***

The value of this variable must be a type specifier symbol for a specific floating-point format; these include `short-float`, `single-float`, `double-float`, and `long-float` and may include implementation-specific types as well. The default value is `single-float`.

`*read-default-float-format*` indicates the floating-point format to be used for reading floating-point numbers that have no exponent marker or have `e` or `E` for an exponent marker. (Other exponent markers explicitly prescribe

the floating-point format to be used.) The printer also uses this variable to guide the choice of exponent markers when printing floating-point numbers.

[Function] **read-preserving-whitespace &optional** *in-stream eof-error-p eof-value recursive-p*

Certain printed representations given to `read`, notably those of symbols and numbers, require a delimiting character after them. (Lists do not, because the close parenthesis marks the end of the list.) Normally `read` will throw away the delimiting character if it is a whitespace character; but `read` will preserve the character (using `unread-char`) if it is syntactically meaningful, because it may be the start of the next expression.

X3J13 voted in January 1989 to clarify the interaction of `unread-char` with echo streams. These changes indirectly affect the echoing behavior of `read-preserving-whitespace`.

The function `read-preserving-whitespace` is provided for some specialized situations where it is desirable to determine precisely what character terminated the extended token.

As an example, consider this macro-character definition:

```
(defun slash-reader (stream char)
  (declare (ignore char))
  (do ((path (list (read-preserving-whitespace stream))
             (cons (progn (read-char stream nil nil t)
                          (read-preserving-whitespace
                            stream))
                   path)))
      ((not (char= (peek-char nil stream nil nil t) #\/))
       (cons 'path (nreverse path)))))
(set-macro-character #\/ #'slash-reader)
```

(This is actually a rather dangerous definition to make because expressions such as `(/ x 3)` will no longer be read properly. The ability to reprogram the reader syntax is very powerful and must be used with caution. This redefinition of `/` is shown here purely for the sake of example.)

Consider now calling `read` on this expression:

```
(zyedh /usr/games/zork /usr/games/boggle)
```

The `/` macro reads objects separated by more `/` characters; thus `/usr/games/zork` is intended to be read as `(path usr games zork)`. The entire example expression should therefore be read as

(zyedh (path usr games zork) (path usr games boggle))

However, if `read` had been used instead of `read-preserving-whitespace`, then after the reading of the symbol `zork`, the following space would be discarded; the next call to `peek-char` would see the following `/`, and the loop would continue, producing this interpretation:

(zyedh (path usr games zork usr games boggle))

On the other hand, there are times when whitespace *should* be discarded. If a command interpreter takes single-character commands, but occasionally reads a Lisp object, then if the whitespace after a symbol is not discarded it might be interpreted as a command some time later after the symbol had been read.

Note that `read-preserving-whitespace` behaves *exactly* like `read` when the *recursive-p* argument is not `nil`. The distinction is established only by calls with *recursive-p* equal to `nil` or omitted.

[Function] **read-delimited-list** *char* **&optional** *input-stream recursive-p*

This reads objects from *stream* until the next character after an object's representation (ignoring whitespace characters and comments) is *char*. (The *char* should not have whitespace syntax in the current readtable.) A list of the objects read is returned.

To be more precise, `read-delimited-list` looks ahead at each step for the next non-whitespace character and peeks at it as if with `peek-char`. If it is *char*, then the character is consumed and the list of objects is returned. If it is a constituent or escape character, then `read` is used to read an object, which is added to the end of the list. If it is a macro character, the associated macro function is called; if the function returns a value, that value is added to the list. The peek-ahead process is then repeated.

X3J13 voted in January 1989 to clarify the interaction of `peek-char` with echo streams. These changes indirectly affect the echoing behavior of the function `read-delimited-list`.

This function is particularly useful for defining new macro characters. Usually it is desirable for the terminating character *char* to be a terminating macro character so that it may be used to delimit tokens; however, `read-delimited-list` makes no attempt to alter the syntax specified for *char* by the current readtable. The user must make any necessary changes to the readtable syntax explicitly. The following example illustrates this.

Suppose you wanted `#{a b c ... z}` to be read as a list of all pairs of the elements *a*, *b*, *c*, ..., *z*; for example:

#{p q z a} reads as ((p q) (p z) (p a) (q z) (q a) (z a))

This can be done by specifying a macro-character definition for `#{` that does two things: read in all the items up to the `}`, and construct the pairs. `read-delimited-list` performs the first task.

Note that `mapcon` allows the mapped function to examine the items of the list after the current one, and that `mapcon` uses `nconc`, which is all right because `mapcar` will produce fresh lists.

```
(defun |#{-reader| (stream char arg)
  (declare (ignore char arg))
  (mapcon #'(lambda (x)
              (mapcar #'(lambda (y) (list (car x) y)) (cdr x)))
          (read-delimited-list #\} stream t)))

(set-dispatch-macro-character #\# #\{ #'|#{-reader|)

(set-macro-character #\} (get-macro-character #\) nil))
```

(Note that `t` is specified for the *recursive-p* argument.)

It is necessary here to give a definition to the character `}` as well to prevent it from being a constituent. If the line

```
(set-macro-character #\} (get-macro-character #\) nil))
```

shown above were not included, then the `}` in

```
#{p q z a}
```

would be considered a constituent character, part of the symbol named `a}`. One could correct for this by putting a space before the `}`, but it is better simply to use the call to `set-macro-character`.

Giving `}` the same definition as the standard definition of the character `)` has the twin benefit of making it terminate tokens for use with `read-delimited-list` and also making it illegal for use in any other context (that is, attempting to read a stray `}` will signal an error).

Note that `read-delimited-list` does not take an *eof-error-p* (or *eof-value*) argument. The reason is that it is always an error to hit end-of-file during the operation of `read-delimited-list`.

[Function] **read-line** **&optional** *input-stream eof-error-p eof-value recursive-p*

`read-line` reads in a line of text terminated by a newline. It returns the line as a character string (*without* the newline character). This function is

usually used to get a line of input from the user. A second returned value is a flag that is false if the line was terminated normally, or true if end-of-file terminated the (non-empty) line. If end-of-file is encountered immediately (that is, appears to terminate an empty line), then end-of-file processing is controlled in the usual way by the *eof-error-p*, *eof-value*, and *recursive-p* arguments.

The corresponding output function is `write-line`.

[Function] **read-char** **&optional** *input-stream eof-error-p eof-value recursive-p*

`read-char` inputs one character from *input-stream* and returns it as a character object.

The corresponding output function is `write-char`.

X3J13 voted in January 1989 to clarify the interaction of `read-char` with echo streams (as created by `make-echo-stream`). A character is echoed from the input stream to the associated output stream the first time it is seen. If a character is read again because of an intervening `unread-char` operation, the character is not echoed again when read for the second time or any subsequent time.

[Function] **unread-char** *character* **&optional** *input-stream*

`unread-char` puts the *character* onto the front of *input-stream*. The *character* must be the same character that was most recently read from the *input-stream*. The *input-stream* "backs up" over this character; when a character is next read from *input-stream*, it will be the specified character followed by the previous contents of *input-stream*. `unread-char` returns `nil`.

One may apply `unread-char` only to the character most recently read from *input-stream*. Moreover, one may not invoke `unread-char` twice consecutively without an intervening `read-char` operation. The result is that one may back up only by one character, and one may not insert any characters into the input stream that were not already there.

X3J13 voted in January 1989 to clarify that one also may not invoke `unread-char` after invoking `peek-char` without an intervening `read-char` operation. This is consistent with the notion that `peek-char` behaves much like `read-char` followed by `unread-char`.

Обоснование: This is not intended to be a general mechanism, but rather an efficient mechanism for allowing the Lisp reader and other parsers to perform one-

character lookahead in the input stream. This protocol admits a wide variety of efficient implementations, such as simply decrementing a buffer pointer. To have to specify the character in the call to `unread-char` is admittedly redundant, since at any given time there is only one character that may be legally specified. The redundancy is intentional, again to give the implementation latitude.

X3J13 voted in January 1989 to clarify the interaction of `unread-char` with echo streams (as created by `make-echo-stream`). When a character is "unread" from an echo stream, no attempt is made to "unecho" the character. However, a character placed back into an echo stream by `unread-char` will not be re-echoed when it is subsequently re-read by `read-char`.

[Function] **peek-char &optional** *peek-type input-stream eof-error-p eof-value recursive-p*

What `peek-char` does depends on the *peek-type*, which defaults to `nil`. With a *peek-type* of `nil`, `peek-char` returns the next character to be read from *input-stream*, without actually removing it from the input stream. The next time input is done from *input-stream*, the character will still be there. It is as if one had called `read-char` and then `unread-char` in succession.

If *peek-type* is `t`, then `peek-char` skips over whitespace characters (but not comments) and then performs the peeking operation on the next character. This is useful for finding the (possible) beginning of the next printed representation of a Lisp object. The last character examined (the one that starts an object) is not removed from the input stream.

If *peek-type* is a character object, then `peek-char` skips over input characters until a character that is `char=` to that object is found; that character is left in the input stream.

X3J13 voted in January 1989 to clarify the interaction of `peek-char` with echo streams (as created by `make-echo-stream`). When a character from an echo stream is only peeked at, it is not echoed at that time. The character remains in the input stream and may be echoed when read by `read-char` at a later time. Note, however, that if the *peek-type* is not `nil`, then characters skipped over (and therefore consumed) by `peek-char` are treated as if they had been read by `read-char`, and will be echoed if `read-char` would have echoed them.

[Function] **listen &optional** *input-stream*

The predicate `listen` is true if there is a character immediately available from *input-stream*, and is false if not. This is particularly useful when the stream obtains characters from an interactive device such as a keyboard. A call to `read-char` would simply wait until a character was available, but `listen` can sense whether or not input is available and allow the program to decide whether or not to attempt input. On a non-interactive stream, the general rule is that `listen` is true except when at end-of-file.

[Function] **read-char-no-hang &optional** *input-stream eof-error-p eof-value recursive-p*

This function is exactly like `read-char`, except that if it would be necessary to wait in order to get a character (as from a keyboard), `nil` is im-

mediately returned without waiting. This allows one to efficiently check for input availability and get the input if it is available. This is different from the `listen` operation in two ways. First, `read-char-no-hang` potentially reads a character, whereas `listen` never inputs a character. Second, `listen` does not distinguish between end-of-file and no input being available, whereas `read-char-no-hang` does make that distinction, returning *eof-value* at end-of-file (or signaling an error if no *eof-error-p* is true) but always returning `nil` if no input is available.

[Function] **clear-input** **&optional** *input-stream*

This clears any buffered input associated with *input-stream*. It is primarily useful for clearing type-ahead from keyboards when some kind of asynchronous error has occurred. If this operation doesn't make sense for the stream involved, then `clear-input` does nothing. `clear-input` returns `nil`.

[Function] **read-from-string** *string* **&optional** *eof-error-p eof-value* **&key** *:start :end :preserve-whitespace*

The characters of *string* are given successively to the Lisp reader, and the Lisp object built by the reader is returned. Macro characters and so on will all take effect.

The arguments `:start` and `:end` delimit a substring of *string* beginning at the character indexed by `:start` and up to but not including the character indexed by `:end`. By default `:start` is `0` (the beginning of the string) and `:end` is `(length` *`string`*`)`. This is the same as for other string functions.

The flag `:preserve-whitespace`, if provided and not `nil`, indicates that the operation should preserve whitespace as for `read-preserving-whitespace`. It defaults to `nil`.

As with other reading functions, the arguments *eof-error-p* and *eof-value* control the action if the end of the (sub)string is reached before the operation is completed; reaching the end of the string is treated as any other end-of-file event.

`read-from-string` returns two values: the first is the object read, and the second is the index of the first character in the string not read. If the entire string was read, the second result will be either the length of the string or one greater than the length of the string. The parameter `:preserve-whitespace` may affect this second value.

(read-from-string "(a b c)") ⇒ (a b c) and 7

[Function] **parse-integer** *string* **&key** *:start :end :radix :junk-allowed*

This function examines the substring of *string* delimited by `:start` and `:end` (which default to the beginning and end of the string). It skips over whitespace characters and then attempts to parse an integer. The `:radix` parameter defaults to `10` and must be an integer between 2 and 36.

If `:junk-allowed` is not `nil`, then the first value returned is the value of the number parsed as an integer or `nil` if no syntactically correct integer was seen.

If `:junk-allowed` is `nil` (the default), then the entire substring is scanned. The returned value is the value of the number parsed as an integer. An error is signaled if the substring does not consist entirely of the representation of an integer, possibly surrounded on either side by whitespace characters.

In either case, the second value is the index into the string of the delimiter that terminated the parse, or it is the index beyond the substring if the parse terminated at the end of the substring (as will always be the case if `:junk-allowed` is false).

Note that `parse-integer` does not recognize the syntactic radix-specifier prefixes `#O`, `#B`, `#X`, and `#`*n*`R`, nor does it recognize a trailing decimal point. It permits only an optional sign (`+` or `-`) followed by a non-empty sequence of digits in the specified radix.

[Function] **read-sequence** *sequence input-stream* **&key** *:start :end*

This function reads elements from input-stream into sequence. The position of the first unchanged element of sequence is returned.

22.2.2 Input from Binary Streams

Common Lisp currently specifies only a very simple facility for binary input: the reading of a single byte as an integer.

[Function] **read-byte** *binary-input-stream* **&optional** *eof-error-p eof-value*

`read-byte` reads one byte from the *binary-input-stream* and returns it in the form of an integer.

22.3 Output Functions

The output functions are divided into two groups: those that operate on streams of characters and those that operate on streams of binary data. The function `format` operates on streams of characters but is described in a section separate from the other character-output functions because of its great complexity.

22.3.1 Output to Character Streams

These functions all take an optional argument called *output-stream*, which is where to send the output. If unsupplied or `nil`, *output-stream* defaults to the value of the variable `*standard-output*`. If it is `t`, the value of the variable `*terminal-io*` is used.

X3J13 voted in June 1989 to add the keyword argument `:readably` to the function `write`, and voted in June 1989 to add the keyword arguments `:right-margin`, `:miser-width`, `:lines`, and `:pprint-dispatch`. The revised description is as follows.

[Function] **write** *object* **&key** *:stream :escape :radix :base :circle :pretty :level :length :case :gensym :array :readably :right-margin :miser-width :lines :pprint-dispatch*

The printed representation of *object* is written to the output stream specified by `:stream`, which defaults to the value of `*standard-output*`.

The other keyword arguments specify values used to control the generation of the printed representation. Each defaults to the value of the corresponding global variable: see `*print-escape*`, `*print-radix*`, `*print-base*`, `*print-circle*`, `*print-pretty*`, `*print-level*`, `*print-length*`, and `*print-case*`, in addition to `*print-array*`,

`*print-gensym*`, `*print-readably*`, `*print-right-margin*`, `*print-miser-width*`, `*print-lines*`, and `*print-pprint-dispatch*`. (This is the means by which these variables affect printing operations: supplying default values for the `write` function.) Note that the printing of symbols is also affected by the value of the variable `*package*`. `write` returns *object*.

[Function] **prin1** *object* **&optional** *output-stream*
[Function] **print** *object* **&optional** *output-stream*
[Function] **pprint** *object* **&optional** *output-stream*
[Function] **princ** *object* **&optional** *output-stream*

`prin1` outputs the printed representation of *object* to *output-stream*. Escape characters are used as appropriate. Roughly speaking, the output from `prin1` is suitable for input to the function `read`. `prin1` returns the *object* as its value.

(prin1 *object output-stream*)
≡ (write *object* :stream *output-stream* :escape t)

`print` is just like `prin1` except that the printed representation of *object* is preceded by a newline (see `terpri`) and followed by a space. `print` returns *object*.

`pprint` is just like `print` except that the trailing space is omitted and the *object* is printed with the `*print-pretty*` flag non-`nil` to produce "pretty" output. `pprint` returns no values (that is, what the expression `(values)` returns: zero values).

X3J13 voted in January 1989 to adopt a facility for user-controlled pretty printing (see chapter 26).

`princ` is just like `prin1` except that the output has no escape characters. A symbol is printed as simply the characters of its print name; a string is printed without surrounding double quotes; and there may be differences for other data types as well. The general rule is that output from `princ` is intended to look good to people, while output from `prin1` is intended to be acceptable to the function `read`. X3J13 voted in June 1987 to clarify that `princ` prints a character in exactly the same manner as `write-char`: the character is simply sent to the output *stream*. This was implied by the

specification in section 22.1.6 in the first edition, but is worth pointing out explicitly here. `princ` returns the *object* as its value.

(princ *object output-stream*)
≡ (write *object* :stream *output-stream* :escape `nil`)

[Function] **write-to-string** *object* **&key** *:escape :radix :base :circle :pretty :level :length :case :gensym :array :readably :right-margin :miser-width :lines :pprint-dispatch*
[Function] **prin1-to-string** *object*
[Function] **princ-to-string** *object*

The object is effectively printed as if by `write`, `prin1`, or `princ`, respectively, and the characters that would be output are made into a string, which is returned.

[Function] **write-char** *character* **&optional** *output-stream*

`write-char` outputs the *character* to *output-stream*, and returns *character*.

[Function] **write-string** *string* **&optional** *output-stream* **&key** *:start :end*
[Function] **write-line** *string* **&optional** *output-stream* **&key** *:start :end*

`write-string` writes the characters of the specified substring of *string* to the *output-stream*. The `:start` and `:end` parameters delimit a substring of *string* in the usual manner (see chapter 14). `write-line` does the same thing but then outputs a newline afterwards. (See `read-line`.) In either case, the *string* is returned (*not* the substring delimited by `:start` and `:end`). In some implementations these may be much more efficient than an explicit loop using `write-char`.

[Function] **write-sequence** *sequence output-stream* **&key** *:start :end*

write-sequence writes the elements of the subsequence of *sequence* bounded by *start* and *end* to *output-stream*.

[Function] **terpri** **&optional** *output-stream*
[Function] **fresh-line** **&optional** *output-stream*

The function `terpri` outputs a newline to *output-stream*. It is identical in effect to `(write-char #\Newline output-stream)`; however, `terpri` always returns `nil`.

`fresh-line` is similar to `terpri` but outputs a newline only if the stream is not already at the start of a line. (If for some reason this cannot be determined, then a newline is output anyway.) This guarantees that the stream will be on a "fresh line" while consuming as little vertical distance as possible. `fresh-line` is a predicate that is true if it output a newline, and otherwise false.

[Function] **finish-output** **&optional** *output-stream*
[Function] **force-output** **&optional** *output-stream*
[Function] **clear-output** **&optional** *output-stream*

Some streams may be implemented in an asynchronous or buffered manner. The function `finish-output` attempts to ensure that all output sent to *output-stream* has reached its destination, and only then returns `nil`. `force-output` initiates the emptying of any internal buffers but returns `nil` without waiting for completion or acknowledgment.

The function `clear-output`, on the other hand, attempts to abort any outstanding output operation in progress in order to allow as little output as possible to continue to the destination. This is useful, for example, to abort a lengthy output to the terminal when an asynchronous error occurs. `clear-output` returns `nil`.

The precise actions of all three of these operations are implementation-dependent.

[Макрос] **print-unreadable-object** (object stream
[[`:type` type | `:identity` id]])
{declaration}* {form}*

Function will output a printed representation of *object* on *stream*, beginning with `#<` and ending with `>`. Everything output to the *stream* during execution of the body forms is enclosed in the angle brackets. If *type* is true, the body output is preceded by a brief description of the object's type and a space character. If *id* is true, the body output is followed by a space character and a representation of the object's identity, typically a storage address.

If `*print-readably*` is true, `print-unreadable-object` signals an error of type `print-not-readable` without printing anything.

The *object*, *stream*, *type*, and *id* arguments are all evaluated normally. The *type* and *id* default to false. It is valid to provide no body forms. If *type* and *id* are both true and there are no body forms, only one space character separates the printed type and the printed identity.

The value returned by `print-unreadable-object` is `nil`.

```
(defmethod print-object ((obj airplane) stream)
  (print-unreadable-object (obj stream :type t :identity t)
    (princ (tail-number obj) stream)))
(print my-airplane)  prints
#<Airplane NW0773 777500123135>      ;In implementation A
                or perhaps
#<FAA:AIRPLANE NW0773 17>            ;In implementation B
```

The big advantage of `print-unreadable-object` is that it allows a user to write `print-object` methods that adhere to implementation-specific style without requiring the user to write implementation-dependent code.

The X3J13 vote left it unclear whether `print-unreadable-object` permits declarations to appear before the body of the macro call. I believe that was the intent, and this is reflected in the syntax shown above; but this is only my interpretation.

22.3.2 Output to Binary Streams

Common Lisp currently specifies only a very simple facility for binary output: the writing of a single byte as an integer.

[Function] **write-byte** *integer binary-output-stream*

`write-byte` writes one byte, the value of *integer*. It is an error if *integer* is not of the type specified as the `:element-type` argument to `open` when the stream was created. The value *integer* is returned.

22.3.3 Formatted Output to Character Streams

The function `format` is very useful for producing nicely formatted text, producing good-looking messages, and so on. `format` can generate a string or output to a stream.

Formatted output is performed not only by the `format` function itself but by certain other functions that accept a control string "the way `format` does." For example, error-signaling functions such as `cerror` accept `format` control strings.

[Function] **format** *destination control-string* **&rest** *arguments*

`format` is used to produce formatted output. `format` outputs the characters of *control-string*, except that a tilde (`~`) introduces a directive. The character after the tilde, possibly preceded by prefix parameters and modifiers, specifies what kind of formatting is desired. Most directives use one or more elements of *arguments* to create their output; the typical directive puts the next element of *arguments* into the output, formatted in some special way. It is an error if no argument remains for a directive requiring an argument, but it is not an error if one or more arguments remain unprocessed by a directive.

The output is sent to *destination*. If *destination* is `nil`, a string is created that contains the output; this string is returned as the value of the call to `format`.

When the first argument to `format` is `nil`, `format` creates a stream of type `string-stream` in much the same manner as `with-output-to-string`. (This stream may be visible to the user if, for example, the `~S` directive is used to print a `defstruct` structure that has a user-supplied print function.)

In all other cases `format` returns `nil`, performing output to *destination* as a side effect. If *destination* is a stream, the output is sent to it. If *destination* is `t`, the output is sent to the stream that is the value of the variable `*standard-output*`. If *destination* is a string with a fill pointer, then in effect the output characters are added to the end of the string (as if by use of `vector-push-extend`).

The `format` function includes some extremely complicated and specialized features. It is not necessary to understand all or even most of its features to use `format` effectively. The beginner should skip over anything in the following documentation that is not immediately useful or clear. The more sophisticated features (such as conditionals and iteration) are there for

the convenience of programs with especially complicated formatting requirements.

A `format` directive consists of a tilde (`~`), optional prefix parameters separated by commas, optional colon (`:`) and at-sign (`@`) modifiers, and a single character indicating what kind of directive this is. The alphabetic case of the directive character is ignored. The prefix parameters are generally integers, notated as optionally signed decimal numbers.

If both colon and at-sign modifiers are present, they may appear in either order; thus `~:@R` and `~@:R` mean the same thing. However, it is traditional to put the colon first, and all the examples in this book put colons before at-signs.

Examples of control strings:

```
"~S"          ;An ~S directive with no parameters or modifiers
"~3,-4:@s"    ;An ~S directive with two parameters, 3 and −4,
              ; and both the colon and at-sign flags
"~,+4S"       ;First prefix parameter is omitted and takes
              ; on its default value; the second parameter is 4
```

Sometimes a prefix parameter is used to specify a character, for instance the padding character in a right- or left-justifying operation. In this case a single quote (`'`) followed by the desired character may be used as a prefix parameter, to mean the character object that is the character following the single quote. For example, you can use `~5,'0d` to print an integer in decimal radix in five columns with leading zeros, or `~5,'*d` to get leading asterisks.

In place of a prefix parameter to a directive, you can put the letter `V` (or `v`), which takes an argument from *arguments* for use as a parameter to the directive. Normally this should be an integer or character object, as appropriate. This feature allows variable-width fields and the like. If the argument used by a `V` parameter is `nil`, the effect is as if the parameter had been omitted. You may also use the character `#` in place of a parameter; it represents the number of arguments remaining to be processed.

It is an error to give a format directive more parameters than it is described here as accepting. It is also an error to give colon or at-sign modifiers to a directive in a combination not specifically described here as being meaningful.

X3J13 voted in January 1989 to clarify the interaction between `format` and the various printer control variables (those named `*print-`*xxx*`*`). This is important because many `format` operations are defined, directly or indirectly, in terms of `prin1` or `princ`, which are affected by the printer control variables. The general rule is that `format` does not bind any of the standard printer control variables except as specified in the individual descriptions of directives. An implementation may not bind any standard printer control variable not specified in the description of a `format` directive, nor may an implementation fail to bind any standard printer control variables that is specified to be bound by such a description. (See these descriptions for specific changes voted by X3J13.)

One consequence of this change is that the user is guaranteed to be able to use the `format` `~A` and `~S` directives to do pretty printing, under control of the `*print-pretty*` variable. Implementations have differed on this point in their interpretations of the first edition. The new `~W` directive may be more appropriate than either `~A` and `~S` for some purposes, whether for pretty printing or ordinary printing. See section 26.4 for a discussion of `~W` and other new `format` directives related to pretty printing.

Here are some relatively simple examples to give you the general flavor of how `format` is used.

(format `nil` "foo") ⇒ "foo"

(setq x 5)

(format `nil` "The answer is ~D." x) ⇒ "The answer is 5."

(format `nil` "The answer is ~3D." x) ⇒ "The answer is   5."

(format `nil` "The answer is ~3,'0D." x) ⇒ "The answer is 005."

(format `nil` "The answer is ~:D." (expt 47 x))
⇒ "The answer is 229,345,007."

(setq y "elephant")

(format `nil` "Look at the ~A!" y) ⇒ "Look at the elephant!"

```
(format nil "Type ~:C to ~A."
        (set-char-bit #\D :control t)
        "delete all your files")
  ⇒ "Type Control-D to delete all your files."
```

(setq n 3)

(format `nil` "~D item~:P found." n) ⇒ "3 items found."

(format `nil` "~R dog~:[s are~; is~] here." n (= n 1))
⇒ "three dogs are here."

(format `nil` "~R dog~:*~[s are~; is~:;s are~] here." n)
⇒ "three dogs are here."

(format `nil` "Here ~[are~;is~:;are~] ~:*~R pupp~:@P." n)
⇒ "Here are three puppies."

In the descriptions of the directives that follow, the term *arg* in general refers to the next item of the set of *arguments* to be processed. The word or phrase at the beginning of each description is a mnemonic (not necessarily an accurate one) for the directive.

`~A` *Ascii.* An *arg*, any Lisp object, is printed without escape characters (as by `princ`). In particular, if *arg* is a string, its characters will be output verbatim. If *arg* is `nil`, it will be printed as `nil`; the colon modifier (`~:A`) will cause an *arg* of `nil` to be printed as (), but if *arg* is a composite structure, such as a list or vector, any contained occurrences of `nil` will still be printed as `nil`.

`~`*mincol*`A` inserts spaces on the right, if necessary, to make the width at least *mincol* columns. The `@` modifier causes the spaces to be inserted on the left rather than the right.

`~`*mincol*`,`*colinc*`,`*minpad*`,`*padchar*`A` is the full form of `~A`, which allows elaborate control of the padding. The string is padded on the right (or on the left if the `@` modifier is used) with at least *minpad* copies of *padchar*; padding characters are then inserted *colinc* characters at a time until the total width is at least *mincol*. The defaults are `0` for *mincol* and *minpad*, `1` for *colinc*, and the space character for *padchar*.

`format` binds `*print-escape*` to `nil` during the processing of the `~A` directive.

`~S` *S-expression*. This is just like `~A`, but *arg* is printed *with* escape characters (as by `prin1` rather than `princ`). The output is therefore suitable for input to `read`. `~S` accepts all the arguments and modifiers that `~A` does.

`format` binds `*print-escape*` to `t` during the processing of the `~S` directive.

~D *Decimal.* An *arg*, which should be an integer, is printed in decimal radix. ~D will never put a decimal point after the number.

~*mincol*D uses a column width of *mincol*; spaces are inserted on the left if the number requires fewer than *mincol* columns for its digits and sign. If the number doesn't fit in *mincol* columns, additional columns are used as needed.

~*mincol*,*padchar*D uses *padchar* as the pad character instead of space.

If *arg* is not an integer, it is printed in ~A format and decimal base.

format binds *print-escape* to nil, *print-radix* to nil, and *print-base* to 10 during processing of ~D.

The @ modifier causes the number's sign to be printed always; the default is to print it only if the number is negative. The : modifier causes commas to be printed between groups of three digits; the third prefix parameter may be used to change the character used as the comma. Thus the most general form of ~D is ~*mincol*,*padchar*,*commachar*D.

X3J13 voted in March 1988 to add a fourth parameter, the *commainterval*. This must be an integer; if it is not provided, it defaults to 3. This parameter controls the number of digits in each group separated by the *commachar*.

By extension, each of the ~B, ~O, and ~X directives accepts a *commainterval* as a fourth parameter, and the ~R directive accepts a *commainterval* as its fifth parameter. Examples:

(format nil "~„' ,4B" #xFACE) ⇒ "1111 1010 1100 1110"
(format nil "~„' ,4B" #x1CE) ⇒ "1 1100 1110"
(format nil "~19„' ,4B" #xFACE) ⇒ "1111 1010 1100 1110"
(format nil "~19„' ,4B" #x1CE) ⇒ "0000 0001 1100 1110"

This is one of those little improvements that probably don't matter much but aren't hard to implement either. It was pretty silly having the number 3 wired into the definition of comma separation when it is just as easy to make it a parameter.

~B *Binary.* This is just like ~D but prints in binary radix (radix 2) instead of decimal. The full form is therefore ~*mincol*,*padchar*,*commachar*B.

`format` binds `*print-escape*` to `nil`, `*print-radix*` to `nil`, and `*print-base*` to `2` during processing of `~B`.

`~O` *Octal.* This is just like `~D` but prints in octal radix (radix 8) instead of decimal. The full form is therefore *`~mincol,padchar,commachar`*`O`.

`format` binds `*print-escape*` to `nil`, `*print-radix*` to `nil`, and `*print-base*` to `8` during processing of `~O`.

`~X` *Hexadecimal.* This is just like `~D` but prints in hexadecimal radix (radix 16) instead of decimal. The full form is therefore *`~mincol,padchar,commachar`*`X`.

`format` binds `*print-escape*` to `nil`, `*print-radix*` to `nil`, and `*print-base*` to `16` during processing of `~X`.

`~R` *Radix.* *`~n`*`R` prints *arg* in radix *n*. The modifier flags and any remaining parameters are used as for the `~D` directive. Indeed, `~D` is the same as `~10R`. The full form here is therefore *`~radix,mincol,padchar,commachar`*`R`.

`format` binds `*print-escape*` to `nil`, `*print-radix*` to `nil`, and `*print-base*` to the value of the first parameter during the processing of the `~R` directive with a parameter.

If no parameters are given to `~R`, then an entirely different interpretation is given. *Notice of correction.* In the first edition, this sentence referred to "arguments" given to `~R`. The correct term is "parameters." The argument should be an integer; suppose it is `4`. Then `~R` prints *arg* as a cardinal English number: `four`; `~:R` prints *arg* as an ordinal English number: `fourth`; `~@R` prints *arg* as a Roman numeral: `IV`; and `~:@R` prints *arg* as an old Roman numeral: `IIII`.

`format` binds `*print-base*` to `10` during the processing of the `~R` directive with no parameter.

The first edition did not specify how `~R` and its variants should handle arguments that are very large or not positive. Actual practice varies, and X3J13 has not yet addressed the topic. Here is a sampling of current practice.

For `~@R` and `~:@R`, nearly all implementations produce Roman numerals only for integers in the range 1 to 3999, inclusive. Some implementations will produce old-style Roman numerals for integers in the range

1 to 4999, inclusive. All other integers are printed in decimal notation, as if `~D` had been used.

For zero, most implementations print `zero` for `~R` and `zeroth` for `~:R`.

For `~R` with a negative argument, most implementations simply print the word `minus` followed by its absolute value as a cardinal in English.

For `~:R` with a negative argument, some implementations also print the word `minus` followed by its absolute value as an ordinal in English; other implementations print the absolute value followed by the word `previous`. Thus the argument `-4` might produce `minus fourth` or `fourth previous`. Each has its charm, but one is not always a suitable substitute for the other; users should be careful.

There is standard English nomenclature for fairly large integers (up to $10^{6}0$, at least), based on appending the suffix -illion to Latin names of integers. Thus we have the names *trillion*, *quadrillion*, *sextillion*, *septillion*, and so on. For extremely large integers, one may express powers of ten in English. One implementation gives `1606938044258990275541962092341162602522202993782792835301376` (which is 2^{200}, the result of `(ash 1 200)`) in this manner:

one times ten to the sixtieth power six hundred six times ten to the fifty-seventh power nine hundred thirty-eight septdecillion forty-four sexdecillion two hundred fifty-eight quindecillion nine hundred ninety quattuordecillion two hundred seventy-five tredecillion five hundred forty-one duodecillion nine hundred sixty-two undecillion ninety-two decillion three hundred forty-one nonillion one hundred sixty-two octillion six hundred two septillion five hundred twenty-two sextillion two hundred two quintillion nine hundred ninety-three quadrillion seven hundred eighty-two trillion seven hundred ninety-two billion eight hundred thirty-five million three hundred one thousand three hundred seventy-six

Another implementation prints it this way (note the use of `plus`):

one times ten to the sixtieth power plus six hundred six times ten to the fifty-seventh power plus ... plus two hundred seventy-five times ten to the forty-second power plus five hundred forty-one duodecillion nine hundred sixty-two undecillion ... three hundred seventy-six

(I have elided some of the text here to save space.)

Unfortunately, the meaning of this nomenclature differs between American English (in which k-illion means $10^{3(k+1)}$, so one trillion is 10^{12}) and British English (in which k-illion means 10^{6k}, so one trillion is 10^{18}). To avoid both confusion and prolixity, I recommend using decimal notation for all numbers above 999,999,999; this is similar to the escape hatch used for Roman numerals.

`~P` *Plural.* If *arg* is not `eql` to the integer `1`, a lowercase `s` is printed; if *arg* is `eql` to `1`, nothing is printed. (Notice that if *arg* is a floating-point `1.0`, the `s` *is* printed.) `~:P` does the same thing, after doing a `~:*` to back up one argument; that is, it prints a lowercase `s` if the *last* argument was not `1`. This is useful after printing a number using `~D`. `~@P` prints `y` if the argument is `1`, or `ies` if it is not. `~:@P` does the same thing, but backs up first.

(format `nil` "~D tr~:@P/~D win~:P" 7 1) ⇒ "7 tries/1 win"
(format `nil` "~D tr~:@P/~D win~:P" 1 0) ⇒ "1 try/0 wins"
(format `nil` "~D tr~:@P/~D win~:P" 1 3) ⇒ "1 try/3 wins"

`~C` *Character.* The next *arg* should be a character; it is printed according to the modifier flags.

`~C` prints the character in an implementation-dependent abbreviated format. This format should be culturally compatible with the host environment.

X3J13 voted in June 1987 to specify that `~C` performs exactly the same action as `write-char` if the character to be printed has zero for its bits attributes. X3J13 voted in March 1989 to eliminate the bits and font attributes, replacing them with the notion of implementation-defined attributes. The net effect is that characters whose implementation-defined attributes all have the "standard" values should be printed by `~C` in the same way that `write-char` would print them.

`~:C` spells out the names of the control bits and represents non-printing characters by their names: `Control-Meta-F`, `Control-Return`, `Space`. This is a "pretty" format for printing characters.

`~:@C` prints what `~:C` would, and then if the character requires unusual shift keys on the keyboard to type it, this fact is mentioned: `Control-`∂ `(Top-F)`. This is the format for telling the user about a key he or she is expected to type, in prompts, for instance. The precise output may depend not only on the implementation but on the particular I/O devices in use.

`~@C` prints the character so that the Lisp reader can read it, using `#\` syntax.

X3J13 voted in January 1989 to specify that `format` binds `*print-escape*` to `t` during the processing of the `~@C` directive. Other variants of the `~C` directive do not bind any printer control variables.

Обоснование: In some implementations the `~S` directive would do what `~C` does, but `~C` is compatible with Lisp dialects such as MacLisp that do not have a character data type.

`~F` *Fixed-format floating-point.* The next *arg* is printed as a floating-point number.

The full form is `~`*w*`,`*d*`,`*k*`,`*overflowchar*`,`*padchar*`F`. The parameter *w* is the width of the field to be printed; *d* is the number of digits to print after the decimal point; *k* is a scale factor that defaults to zero.

Exactly *w* characters will be output. First, leading copies of the character *padchar* (which defaults to a space) are printed, if necessary, to pad the field on the left. If the *arg* is negative, then a minus sign is printed; if the *arg* is not negative, then a plus sign is printed if and only if the `@` modifier was specified. Then a sequence of digits, containing a single embedded decimal point, is printed; this represents the magnitude of the value of *arg* times 10^k, rounded to *d* fractional digits. (When rounding up and rounding down would produce printed values equidistant from the scaled value of *arg*, then the implementation is free to use either one. For example, printing the argument `6.375` using the format `~4,2F` may correctly produce either `6.37` or `6.38`.) Leading zeros are not permitted, except that a single zero digit is output before the decimal point if the printed value is less than 1, and this single zero digit is not output after all if $w = d + 1$.

If it is impossible to print the value in the required format in a field of width *w*, then one of two actions is taken. If the parameter *overflowchar*

is specified, then w copies of that parameter are printed instead of the scaled value of *arg*. If the *overflowchar* parameter is omitted, then the scaled value is printed using more than w characters, as many more as may be needed.

If the w parameter is omitted, then the field is of variable width. In effect, a value is chosen for w in such a way that no leading pad characters need to be printed and exactly d characters will follow the decimal point. For example, the directive `~,2F` will print exactly two digits after the decimal point and as many as necessary before the decimal point.

If the parameter d is omitted, then there is no constraint on the number of digits to appear after the decimal point. A value is chosen for d in such a way that as many digits as possible may be printed subject to the width constraint imposed by the parameter w and the constraint that no trailing zero digits may appear in the fraction, except that if the fraction to be printed is zero, then a single zero digit should appear after the decimal point if permitted by the width constraint.

If both w and d are omitted, then the effect is to print the value using ordinary free-format output; `prin1` uses this format for any number whose magnitude is either zero or between 10^{-3} (inclusive) and 10^{7} (exclusive).

If w is omitted, then if the magnitude of *arg* is so large (or, if d is also omitted, so small) that more than 100 digits would have to be printed, then an implementation is free, at its discretion, to print the number using exponential notation instead, as if by the directive `~E` (with all parameters to `~E` defaulted, not taking their values from the `~F` directive).

If *arg* is a rational number, then it is coerced to be a `single-float` and then printed. (Alternatively, an implementation is permitted to process a rational number by any other method that has essentially the same behavior but avoids such hazards as loss of precision or overflow because of the coercion. However, note that if w and d are unspecified and the number has no exact decimal representation, for example `1/3`, some precision cutoff must be chosen by the implementation: only a finite number of digits may be printed.)

If *arg* is a complex number or some non-numeric object, then it is

printed using the format directive ~*w*D, thereby printing it in decimal radix and a minimum field width of *w*. (If it is desired to print each of the real part and imaginary part of a complex number using a ~F directive, then this must be done explicitly with two ~F directives and code to extract the two parts of the complex number.)

X3J13 voted in January 1989 to specify that `format` binds `*print-escape*` to `nil` during the processing of the ~F directive.

```
(defun foo (x)
  (format nil "~6,2F|~6,2,1,'*F|~6,2,,'?F|~6F|~,2F|~F"
          x x x x x x))
(foo 3.14159)  ⇒ "  3.14| 31.42|  3.14|3.1416|3.14|3.14159"
(foo -3.14159) ⇒ " -3.14|-31.42| -3.14|-3.142|-3.14|-3.14159"
(foo 100.0)    ⇒ "100.00|******|100.00| 100.0|100.00|100.0"
(foo 1234.0)   ⇒ "1234.00|******|??????|1234.0|1234.00|1234.0"
(foo 0.006)    ⇒ "  0.01|  0.06|  0.01| 0.006|0.01|0.006"
```

~E *Exponential floating-point.* The next *arg* is printed in exponential notation.

The full form is ~*w*,*d*,*e*,*k*,*overflowchar*,*padchar*,*exponentchar*E. The parameter *w* is the width of the field to be printed; *d* is the number of digits to print after the decimal point; *e* is the number of digits to use when printing the exponent; *k* is a scale factor that defaults to 1 (not zero).

Exactly *w* characters will be output. First, leading copies of the character *padchar* (which defaults to a space) are printed, if necessary, to pad the field on the left. If the *arg* is negative, then a minus sign is printed; if the *arg* is not negative, then a plus sign is printed if and only if the @ modifier was specified. Then a sequence of digits, containing a single embedded decimal point, is printed. The form of this sequence of digits depends on the scale factor *k*. If *k* is zero, then *d* digits are printed after the decimal point, and a single zero digit appears before the decimal point if the total field width will permit it. If *k* is positive, then it must be strictly less than $d+2$; *k* significant digits are printed

before the decimal point, and $d - k + 1$ digits are printed after the decimal point. If k is negative, then it must be strictly greater than $-d$; a single zero digit appears before the decimal point if the total field width will permit it, and after the decimal point are printed first $-k$ zeros and then $d + k$ significant digits. The printed fraction must be properly rounded. (When rounding up and rounding down would produce printed values equidistant from the scaled value of *arg*, then the implementation is free to use either one. For example, printing `637.5` using the format `~8,2E` may correctly produce either `6.37E+02` or `6.38E+02`.)

Following the digit sequence, the exponent is printed. First the character parameter *exponentchar* is printed; if this parameter is omitted, then the exponent marker that `prin1` would use is printed, as determined from the type of the floating-point number and the current value of `*read-default-float-format*`. Next, either a plus sign or a minus sign is printed, followed by e digits representing the power of 10 by which the printed fraction must be multiplied to properly represent the rounded value of *arg*.

If it is impossible to print the value in the required format in a field of width w, possibly because k is too large or too small or because the exponent cannot be printed in e character positions, then one of two actions is taken. If the parameter *overflowchar* is specified, then w copies of that parameter are printed instead of the scaled value of *arg*. If the *overflowchar* parameter is omitted, then the scaled value is printed using more than w characters, as many more as may be needed; if the problem is that d is too small for the specified k or that e is too small, then a larger value is used for d or e as may be needed.

If the w parameter is omitted, then the field is of variable width. In effect a value is chosen for w in such a way that no leading pad characters need to be printed.

If the parameter d is omitted, then there is no constraint on the number of digits to appear. A value is chosen for d in such a way that as many digits as possible may be printed subject to the width constraint imposed by the parameter w, the constraint of the scale factor k, and the constraint that no trailing zero digits may appear in the fraction, except that if the fraction to be printed is zero, then a single zero digit

should appear after the decimal point if the width constraint allows it.

If the parameter *e* is omitted, then the exponent is printed using the smallest number of digits necessary to represent its value.

If all of *w*, *d*, and *e* are omitted, then the effect is to print the value using ordinary free-format exponential-notation output; `prin1` uses this format for any non-zero number whose magnitude is less than 10^{-3} or greater than or equal to 10^{7}.

X3J13 voted in January 1989 to amend the previous paragraph as follows:

If all of *w*, *d*, and *e* are omitted, then the effect is to print the value using ordinary free-format exponential-notation output; `prin1` uses a similar format for any non-zero number whose magnitude is less than 10^{-3} or greater than or equal to 10^{7}. The only difference is that the `~E` directive always prints a plus or minus sign before the exponent, while `prin1` omits the plus sign if the exponent is non-negative.

(The amendment reconciles this paragraph with the specification several paragraphs above that `~E` always prints a plus or minus sign before the exponent.)

If *arg* is a rational number, then it is coerced to be a `single-float` and then printed. (Alternatively, an implementation is permitted to process a rational number by any other method that has essentially the same behavior but avoids such hazards as loss of precision or overflow because of the coercion. However, note that if *w* and *d* are unspecified and the number has no exact decimal representation, for example `1/3`, some precision cutoff must be chosen by the implementation: only a finite number of digits may be printed.)

If *arg* is a complex number or some non-numeric object, then it is printed using the format directive `~`*w*`D`, thereby printing it in decimal radix and a minimum field width of *w*. (If it is desired to print each of the real part and imaginary part of a complex number using a `~E` directive, then this must be done explicitly with two `~E` directives and code to extract the two parts of the complex number.)

X3J13 voted in January 1989 to specify that `format` binds `*print-escape*` to `nil` during the processing of the `~E` directive.

```
(defun foo (x)
  (format nil
          "~9,2,1,,'*E|~10,3,2,2,'?,,'$E|~9,3,2,-2,'%@E|~9,2E"
          x x x x))
(foo 3.14159)  ⇒ "  3.14E+0| 31.42$-01|+.003E+03|  3.14E+0"
(foo -3.14159) ⇒ " -3.14E+0|-31.42$-01|-.003E+03| -3.14E+0"
(foo 1100.0)   ⇒ "  1.10E+3| 11.00$+02|+.001E+06|  1.10E+3"
(foo 1100.0L0) ⇒ "  1.10L+3| 11.00$+02|+.001L+06|  1.10L+3"
(foo 1.1E13)   ⇒ "*********| 11.00$+12|+.001E+16| 1.10E+13"
(foo 1.1L120)  ⇒ "*********|??????????|%%%%%%%%%|1.10L+120"
(foo 1.1L1200) ⇒ "*********|??????????|%%%%%%%%%|1.10L+1200"
```

Here is an example of the effects of varying the scale factor:

```
(dotimes (k 13)
  (format t " %Scale factor  2D: | 13,6,2,VE|"
          (- k 5) 3.14159))                              ;Prints 13 lines
Scale factor -5: | 0.000003E+06|
Scale factor -4: | 0.000031E+05|
Scale factor -3: | 0.000314E+04|
Scale factor -2: | 0.003142E+03|
Scale factor -1: | 0.031416E+02|
Scale factor  0: | 0.314159E+01|
Scale factor  1: | 3.141590E+00|
Scale factor  2: | 31.41590E-01|
Scale factor  3: | 314.1590E-02|
Scale factor  4: | 3141.590E-03|
Scale factor  5: | 31415.90E-04|
Scale factor  6: | 314159.0E-05|
Scale factor  7: | 3141590.E-06|
```

`~G` *General floating-point.* The next *arg* is printed as a floating-point number in either fixed-format or exponential notation as appropriate.

The full form is `~`*w*`,`*d*`,`*e*`,`*k*`,`*overflowchar*`,`*padchar*`,`*exponentchar*`G`. The format in which to print *arg* depends on the magnitude (absolute value) of the *arg*. Let n be an integer such that $10^{n-1} \leq arg < 10^n$. (If *arg* is zero, let n be 0.) Let ee equal $e+2$, or 4 if e is omitted. Let ww equal $w-ee$, or `nil` if w is omitted. If d is omitted, first let q be the number of digits needed to print *arg* with no loss of information and without leading or trailing zeros; then let d equal `(max` *q* `(min` *n* `7))`. Let dd equal $d-n$.

If $0 \leq dd \leq d$, then *arg* is printed as if by the format directives

~*ww*,*dd*,,*overflowchar*,*padchar*F~*ee*@T

Note that the scale factor k is not passed to the `~F` directive. For all other values of dd, *arg* is printed as if by the format directive

~*w*,*d*,*e*,*k*,*overflowchar*,*padchar*,*exponentchar*E

In either case, an `@` modifier is specified to the `~F` or `~E` directive if and only if one was specified to the `~G` directive.

`format` binds `*print-escape*` to `nil` during the processing of the `~G` directive.

Examples:

```
(defun foo (x)
  (format nil
          "~9,2,1,,'*G|~9,3,2,3,'?,,'$G|~9,3,2,0,'%G|~9,2G"
          x x x))

(foo 0.0314159) ⇒ "  3.14E-2|314.2$-04|0.314E-01|  3.14E-2"
(foo 0.314159)  ⇒ "  0.31   |0.314    |0.314    | 0.31    "
(foo 3.14159)   ⇒ "   3.1   | 3.14    | 3.14    |  3.1    "
(foo 31.4159)   ⇒ "   31.   | 31.4    | 31.4    |  31.    "
(foo 314.159)   ⇒ "  3.14E+2| 314.    | 314.    |  3.14E+2"
```

```
(foo 3141.59)   ⇒ "  3.14E+3|314.2$+01|0.314E+04|  3.14E+3"
(foo 3141.59L0) ⇒ "  3.14L+3|314.2$+01|0.314L+04|  3.14L+3"
(foo 3.14E12)   ⇒ "*********|314.0$+10|0.314E+13| 3.14E+12"
(foo 3.14L120)  ⇒ "*********|?????????|%%%%%%%%|3.14L+120"
(foo 3.14L1200) ⇒ "*********|?????????|%%%%%%%%|3.14L+1200"
```

`~$` *Dollars floating-point.* The next *arg* is printed as a floating-point number in fixed-format notation. This format is particularly convenient for printing a value as dollars and cents.

The full form is `~`*d*`,`*n*`,`*w*`,`*padchar*`$`. The parameter *d* is the number of digits to print after the decimal point (default value 2); *n* is the minimum number of digits to print before the decimal point (default value 1); *w* is the minimum total width of the field to be printed (default value 0).

First padding and the sign are output. If the *arg* is negative, then a minus sign is printed; if the *arg* is not negative, then a plus sign is printed if and only if the `@` modifier was specified. If the `:` modifier is used, the sign appears before any padding, and otherwise after the padding. If *w* is specified and the number of other characters to be output is less than *w*, then copies of *padchar* (which defaults to a space) are output to make the total field width equal *w*. Then *n* digits are printed for the integer part of *arg*, with leading zeros if necessary; then a decimal point; then *d* digits of fraction, properly rounded.

If the magnitude of *arg* is so large that more than *m* digits would have to be printed, where *m* is the larger of *w* and 100, then an implementation is free, at its discretion, to print the number using exponential notation instead, as if by the directive `~`*w*`,`*q*`,,,`*padchar*`E`, where *w* and *padchar* are present or omitted according to whether they were present or omitted in the `~$` directive, and where $q = d + n - 1$, where *d* and *n* are the (possibly default) values given to the `~$` directive.

If *arg* is a rational number, then it is coerced to be a `single-float` and then printed. (Alternatively, an implementation is permitted to process a rational number by any other method that has essentially the same behavior but avoids such hazards as loss of precision or overflow because of the coercion.)

If *arg* is a complex number or some non-numeric object, then it is printed using the format directive `~`*w*`D`, thereby printing it in decimal radix and a minimum field width of *w*. (If it is desired to print each of the real part and imaginary part of a complex number using a `~$` directive, then this must be done explicitly with two `~$` directives and code to extract the two parts of the complex number.)

`format` binds `*print-escape*` to `nil` during the processing of the `~$`

directive.

`~%` This outputs a `#\Newline` character, thereby terminating the current output line and beginning a new one (see `terpri`).

`~`*n*`%` outputs *n* newlines.

No *arg* is used. Simply putting a newline in the control string would work, but `~%` is often used because it makes the control string look nicer in the middle of a Lisp program.

`~&` Unless it can be determined that the output stream is already at the beginning of a line, this outputs a newline (see `fresh-line`).

`~`*n*`&` calls `fresh-line` and then outputs $n-1$ newlines. `~0&` does nothing.

`~|` This outputs a page separator character, if possible. `~`*n*`|` does this *n* times. `|` is vertical bar, not capital I.

`~~` *Tilde.* This outputs a tilde. `~`*n*`~` outputs *n* tildes.

`~`⟨**newline**⟩ Tilde immediately followed by a newline ignores the newline and any following non-newline whitespace characters. With a `:`, the newline is ignored, but any following whitespace is left in place. With an `@`, the newline is left in place, but any following whitespace is ignored. This directive is typically used when a format control string is too long to fit nicely into one line of the program:

```
(defun type-clash-error (fn nargs argnum right-type wrong-type)
  (format *error-output*
          "~&Function ~S requires its ~:[~:R~;~*~] ~
           argument to be of type ~S,~%but it was called ~
           with an argument of type ~S.~%"
          fn (eql nargs 1) argnum right-type wrong-type))
```

(type-clash-error 'aref nil 2 'integer 'vector) prints:
Function AREF requires its second argument to be of type INTEGER,
but it was called with an argument of type VECTOR.

(type-clash-error 'car 1 1 'list 'short-float) prints:

Function CAR requires its argument to be of type LIST,
but it was called with an argument of type SHORT-FLOAT.

Note that in this example newlines appear in the output only as specified by the `~&` and `~%` directives; the actual newline characters in the control string are suppressed because each is preceded by a tilde.

`~T` *Tabulate.* This spaces over to a given column. `~`*colnum*`,`*colinc*`T` will output sufficient spaces to move the cursor to column *colnum*. If the cursor is already at or beyond column *colnum*, it will output spaces to move it to column *colnum*+*k***colinc* for the smallest positive integer *k* possible, unless *colinc* is zero, in which case no spaces are output if the cursor is already at or beyond column *colnum*. *colnum* and *colinc* default to `1`.

Ideally, the current column position is determined by examination of the destination, whether a stream or string. (Although no user-level operation for determining the column position of a stream is defined by Common Lisp, such a facility may exist at the implementation level.) If for some reason the current absolute column position cannot be determined by direct inquiry, `format` may be able to deduce the current column position by noting that certain directives (such as `~%`, or `~&`, or `~A` with the argument being a string containing a newline) cause the column position to be reset to zero, and counting the number of characters emitted since that point. If that fails, `format` may attempt a similar deduction on the riskier assumption that the destination was at column zero when `format` was invoked. If even this heuristic fails or is implementationally inconvenient, at worst the `~T` operation will simply output two spaces. (All this implies that code that uses `format` is more likely to be portable if all format control strings that use the `~T` directive either begin with `~%` or `~&` to force a newline or are designed to be used only when the destination is known from other considerations to be at column zero.)

`~@T` performs *relative* tabulation. `~`*colrel*`,`*colinc*`@T` outputs *colrel* spaces and then outputs the smallest non-negative number of additional spaces necessary to move the cursor to a column that is a multiple of *colinc*. For example, the directive `~3,8@T` outputs three spaces and then moves the cursor to a "standard multiple-of-eight tab stop" if not

at one already. If the current output column cannot be determined, however, then *colinc* is ignored, and exactly *colrel* spaces are output.

X3J13 voted in June 1989 to define `~:T` and `~:@T` to perform tabulation relative to a point defined by the pretty printing process (see section 26.4).

`~*` The next *arg* is ignored. `~`*n*`*` ignores the next *n* arguments.

`~:*` "ignores backwards"; that is, it backs up in the list of arguments so that the argument last processed will be processed again. `~`*n*`:*` backs up *n* arguments.

When within a `~{` construct (see below), the ignoring (in either direction) is relative to the list of arguments being processed by the iteration.

`~`*n*`@*` is an "absolute goto" rather than a "relative goto": it goes to the *n*th *arg*, where 0 means the first one; *n* defaults to 0, so `~@*` goes back to the first *arg*. Directives after a `~`*n*`@*` will take arguments in sequence beginning with the one gone to. When within a `~{` construct, the "goto" is relative to the list of arguments being processed by the iteration.

`~?` *Indirection.* The next *arg* must be a string, and the one after it a list; both are consumed by the `~?` directive. The string is processed as a `format` control string, with the elements of the list as the arguments. Once the recursive processing of the control string has been finished, then processing of the control string containing the `~?` directive is resumed. Example:

```
(format nil "~? ~D" "<~A ~D>" '("Foo" 5) 7) ⇒ "<Foo 5> 7"
(format nil "~? ~D" "<~A ~D>" '("Foo" 5 14) 7) ⇒ "<Foo 5> 7"
```

Note that in the second example three arguments are supplied to the control string `"<~A ~D>"`, but only two are processed and the third is therefore ignored.

With the `@` modifier, only one *arg* is directly consumed. The *arg* must be a string; it is processed as part of the control string as if it had appeared in place of the `~@?` construct, and any directives in the recursively processed control string may consume arguments of the control string containing the `~@?` directive. Example:

(format nil "~@? ~D" "<~A ~D>" "Foo" 5 7) ⇒ "<Foo 5> 7"
(format nil "~@? ~D" "<~A ~D>" "Foo" 5 14 7) ⇒ "<Foo 5> 14"

Here is a rather sophisticated example. The `format` function itself, as implemented at one time in Lisp Machine Lisp, used a routine internal to the `format` package called `format-error` to signal error messages; `format-error` in turn used `error`, which used `format` recursively. Now `format-error` took a string and arguments, just like `format`, but also printed the control string to `format` (which at this point was available in the global variable `*ctl-string*`) and a little arrow showing where in the processing of the control string the error occurred. The variable `*ctl-index*` pointed one character after the place of the error.

```
(defun format-error (string &rest args)     ;Example
  (error nil "~?~%~V@T⇓~%~3@T\"~A\"~%"
         string args (+ *ctl-index* 3) *ctl-string*))
```

(The character set used in the Lisp Machine Lisp implementation contains a down-arrow character ⇓, which is not a standard Common Lisp character.) This first processed the given string and arguments using `~?`, then output a newline, tabbed a variable amount for printing the down-arrow, and printed the control string between double quotes (note the use of `\"` to include double quotes within the control string). The effect was something like this:

```
(format t "The item is a ~[Foo~;Bar~;Loser~]." 'quux)
»ERROR: The argument to the FORMAT "~[" command
       must be a number.
              ⇓
  "The item is a ~[Foo~;Bar~;Loser~]."
```

Заметка для реализации: Implementors may wish to report errors occurring within `format` control strings in the manner outlined here. It looks pretty flashy when done properly.

X3J13 voted in June 1989 to introduce certain `format` directives to support the user interface to the pretty printer described in detail in chapter 26.

`~_` *Conditional newline.* Without any modifiers, the directive `~_` is equivalent to `(pprint-newline :linear)`. The directive `~@_` is equivalent to `(pprint-newline :miser)`. The directive `~:_` is equivalent to `(pprint-newline :fill)`. The directive `~:@_` is equivalent to `(pprint-newline :mandatory)`.

`~W` *Write.* An *arg*, any Lisp object, is printed obeying *every* printer control variable (as by `write`). See section 26.4 for details.

`~I` *Indent.* The directive `~`*n*`I` is equivalent to `(pprint-indent :block `*n*`)`. The directive `~:`*n*`I` is equivalent to `(pprint-indent :current `*n*`)`. In both cases, *n* defaults to zero, if it is omitted.

The format directives after this point are much more complicated than the foregoing; they constitute control structures that can perform case conversion, conditional selection, iteration, justification, and non-local exits. Used with restraint, they can perform powerful tasks. Used with abandon, they can produce completely unreadable and unmaintainable code.

The case-conversion, conditional, iteration, and justification constructs can contain other formatting constructs by bracketing them. These constructs must nest properly with respect to each other. For example, it is not legitimate to put the start of a case-conversion construct in each arm of a conditional and the end of the case-conversion construct outside the conditional:

```
(format nil "~:[abc~:@(def~;ghi~:@(jkl~]mno~)" x)     ;Illegal!
```

One might expect this to produce either `"abcDEFMNO"` or `"ghiJKLMNO"`, depending on whether `x` is false or true; but in fact the construction is illegal because the `~[...~;...~]` and `~(...~)` constructs are not properly nested.

The processing indirection caused by the `~?` directive is also a kind of nesting for the purposes of this rule of proper nesting. It is not permitted to start a bracketing construct within a string processed under control of a `~?` directive and end the construct at some point after the `~?` construct in the string containing that construct, or vice versa. For example, this situation is illegal:

```
(format nil "~?ghi~)" "abc~@(def")     ;Illegal!
```

One might expect it to produce `"abcDEFGHI"`, but in fact the construction is illegal because the `~?` and `~(...~)` constructs are not properly nested.

`~(`*str*`~)` *Case conversion.* The contained control string *str* is processed, and what it produces is subject to case conversion: `~(` converts every uppercase character to the corresponding lowercase character; `~:(` capitalizes all words, as if by `string-capitalize`; `~@(` capitalizes just the first word and forces the rest to lowercase; `~:@(` converts every lowercase character to the corresponding uppercase character. In this example, `~@(` is used to cause the first word produced by `~@R` to be capitalized:

```
(format nil "~@R ~(~@R~)" 14 14) ⇒ "XIV xiv"
(defun f (n) (format nil "~@(~R~) error~:P detected." n))
(f 0) ⇒ "Zero errors detected."
(f 1) ⇒ "One error detected."
(f 23) ⇒ "Twenty-three errors detected."
```

`~[`*str0*`~;`*str1*`~;...~;`*strn*`~]` *Conditional expression.* This is a set of control strings, called *clauses*, one of which is chosen and used. The clauses are separated by `~;` and the construct is terminated by `~]`. For example,

```
"~[Siamese~;Manx~;Persian~] Cat"
```

The *arg*th clause is selected, where the first clause is number 0. If a prefix parameter is given (as `~`*n*`[`), then the parameter is used instead of an argument. (This is useful only if the parameter is specified by `#`, to dispatch on the number of arguments remaining to be processed.) If *arg* is out of range, then no clause is selected (and no error is signaled). After the selected alternative has been processed, the control string continues after the `~]`.

`~[`*str0*`~;`*str1*`~;...~;`*strn*`~:;`*default*`~]` has a default case. If the *last* `~;` used to separate clauses is `~:;` instead, then the last clause is an "else" clause that is performed if no other clause is selected. For example:

```
"~[Siamese~;Manx~;Persian~:;Alley~] Cat"
```

`~:[`*false*`~;`*true*`~]` selects the *false* control string if *arg* is `nil`, and selects the *true* control string otherwise.

`~@[`*true*`~]` tests the argument. If it is not `nil`, then the argument is not used up by the `~@[` command but remains as the next one to be processed, and the one clause *true* is processed. If the *arg* is `nil`, then the argument is used up, and the clause is not processed. The clause therefore should normally use exactly one argument, and may expect it to be non-`nil`. For example:

```
(setq *print-level* nil *print-length* 5)
(format nil "~@[ print level = ~D~]~@[ print length = ~D~]"
          *print-level* *print-length*)
  ⇒ " print length = 5"
```

The combination of `~[` and `#` is useful, for example, for dealing with English conventions for printing lists:

```
(setq foo "Items:~#[ none~; ~S~; ~S and ~S~
        ~:;~@{~#[~; and~] ~S~^,~}~].")
(format nil foo)
      ⇒ "Items: none."
(format nil foo 'foo)
      ⇒ "Items: FOO."
(format nil foo 'foo 'bar)
      ⇒ "Items: FOO and BAR."
(format nil foo 'foo 'bar 'baz)
      ⇒ "Items: FOO, BAR, and BAZ."
(format nil foo 'foo 'bar 'baz 'quux)
      ⇒ "Items: FOO, BAR, BAZ, and QUUX."
```

`~;` This separates clauses in `~[` and `~<` constructions. It is an error elsewhere.

`~]` This terminates a `~[`. It is an error elsewhere.

`~{`*str*`~}` *Iteration.* This is an iteration construct. The argument should be a list, which is used as a set of arguments as if for a recursive call to `format`. The string *str* is used repeatedly as the control string. Each iteration can absorb as many elements of the list as it likes as arguments; if *str* uses up two arguments by itself, then two elements of the list will get used up each time around the loop. If before any iteration step the list is empty, then the iteration is terminated. Also, if a prefix parameter *n* is given, then there will be at most *n* repetitions of processing of *str*. Finally, the `~^` directive can be used to terminate the iteration prematurely.

Here are some simple examples:

```
(format nil
        "The winners are:~{ ~S~}."
        '(fred harry jill))
    ⇒ "The winners are: FRED HARRY JILL."

(format nil "Pairs:~{ <~S,~S>~}." '(a 1 b 2 c 3))
    ⇒ "Pairs: <A,1> <B,2> <C,3>."
```

`~:{`*str*`~}` is similar, but the argument should be a list of sublists. At each repetition step, one sublist is used as the set of arguments for processing *str*; on the next repetition, a new sublist is used, whether or not all of the last sublist had been processed. Example:

```
(format nil "Pairs:~:{ <~S,~S>~}."
            '((a 1) (b 2) (c 3)))
    ⇒ "Pairs: <A,1> <B,2> <C,3>."
```

`~@{`*str*`~}` is similar to `~{`*str*`~}`, but instead of using one argument that is a list, all the remaining arguments are used as the list of arguments for the iteration. Example:

```
(format nil "Pairs:~@{ <~S,~S>~}."
        'a 1 'b 2 'c 3)
    ⇒ "Pairs: <A,1> <B,2> <C,3>."
```

If the iteration is terminated before all the remaining arguments are consumed, then any arguments not processed by the iteration remain to be processed by any directives following the iteration construct.

`~:@{`*str*`~}` combines the features of `~:{`*str*`~}` and `~@{`*str*`~}`. All the remaining arguments are used, and each one must be a list. On each iteration, the next argument is used as a list of arguments to *str*. Example:

```
(format nil "Pairs:~:@{ <~S,~S>~}."
        '(a 1) '(b 2) '(c 3))
    ⇒ "Pairs: <A,1> <B,2> <C,3>."
```

Terminating the repetition construct with `~:}` instead of `~}` forces *str* to be processed at least once, even if the initial list of arguments is null (however, it will not override an explicit prefix parameter of zero).

If *str* is empty, then an argument is used as *str*. It must be a string and precede any arguments processed by the iteration. As an example, the following are equivalent:

```
(apply #'format stream string arguments)
(format stream "~1{~:}" string arguments)
```

This will use `string` as a formatting string. The `~1{` says it will be processed at most once, and the `~:}` says it will be processed at least once. Therefore it is processed exactly once, using `arguments` as the arguments. This case may be handled more clearly by the `~?` directive, but this general feature of `~{` is more powerful than `~?`.

`~}` This terminates a `~{`. It is an error elsewhere.

`~`*`mincol`*`,`*`colinc`*`,`*`minpad`*`,`*`padchar`*`<`*`str`*`~>` *Justification.* This justifies the text produced by processing *str* within a field at least *mincol* columns wide. *str* may be divided up into segments with `~;`, in which case the spacing is evenly divided between the text segments.

With no modifiers, the leftmost text segment is left-justified in the field, and the rightmost text segment right-justified; if there is only one text element, as a special case, it is right-justified. The `:` modifier causes spacing to be introduced before the first text segment; the `@` modifier causes spacing to be added after the last. The *minpad* parameter (default `0`) is the minimum number of padding characters to be output between each segment. The padding character is specified by *padchar*, which defaults to the space character. If the total width needed to satisfy these constraints is greater than *mincol*, then the width used is *mincol*+*k***colinc* for the smallest possible non-negative integer value *k*; *colinc* defaults to `1`, and *mincol* defaults to `0`.

```
(format nil "~10<foo~;bar~>")        ⇒ "foo    bar"
(format nil "~10:<foo~;bar~>")       ⇒ "  foo  bar"
(format nil "~10:@<foo~;bar~>")       ⇒ "  foo bar "
(format nil "~10<foobar~>")          ⇒ "    foobar"
(format nil "~10:<foobar~>")         ⇒ "    foobar"
(format nil "~10@<foobar~>")          ⇒ "foobar    "
(format nil "~10:@<foobar~>")         ⇒ "  foobar  "
```

Note that *str* may include `format` directives. All the clauses in *str* are processed in order; it is the resulting pieces of text that are justified.

The `~^` directive may be used to terminate processing of the clauses prematurely, in which case only the completely processed clauses are justified.

If the first clause of a `~<` is terminated with `~:;` instead of `~;`, then it is used in a special way. All of the clauses are processed (subject to `~^`, of course), but the first one is not used in performing the spacing and padding. When the padded result has been determined, then if it will fit on the current line of output, it is output, and the text for

the first clause is discarded. If, however, the padded text will not fit on the current line, then the text segment for the first clause is output before the padded text. The first clause ought to contain a newline (such as a `~%` directive). The first clause is always processed, and so any arguments it refers to will be used; the decision is whether to use the resulting segment of text, not whether to process the first clause. If the `~:;` has a prefix parameter *n*, then the padded text must fit on the current line with *n* character positions to spare to avoid outputting the first clause's text. For example, the control string

```
"~%;; ~{~<~%;; ~1:; ~S~>~^,~}.~%"
```

can be used to print a list of items separated by commas without breaking items over line boundaries, beginning each line with `;;` . The prefix parameter `1` in `~1:;` accounts for the width of the comma that will follow the justified item if it is not the last element in the list, or the period if it is. If `~:;` has a second prefix parameter, then it is used as the width of the line, thus overriding the natural line width of the output stream. To make the preceding example use a line width of 50, one would write

```
"~%;; ~{~<~%;; ~1,50:; ~S~>~^,~}.~%"
```

If the second argument is not specified, then `format` uses the line width of the output stream. If this cannot be determined (for example, when producing a string result), then `format` uses `72` as the line length.

`~>` Terminates a `~<`. It is an error elsewhere. X3J13 voted in June 1989 to introduce certain `format` directives to support the user interface to the pretty printer. If `~:>` is used to terminate a `~<...` directive, the directive is equivalent to a call on `pprint-logical-block`. See section 26.4 for details.

`~^` *Up and out.* This is an escape construct. If there are no more arguments remaining to be processed, then the immediately enclosing `~{` or `~<` construct is terminated. If there is no such enclosing construct,

then the entire formatting operation is terminated. In the ~< case, the formatting *is* performed, but no more segments are processed before doing the justification. The ~^ should appear only at the *beginning* of a ~< clause, because it aborts the entire clause it appears in (as well as all following clauses). ~^ may appear anywhere in a ~{ construct.

(setq donestr "Done.~^ ~D warning~:P.~^ ~D error~:P.")
(format nil donestr) ⇒ "Done."
(format nil donestr 3) ⇒ "Done. 3 warnings."
(format nil donestr 1 5) ⇒ "Done. 1 warning. 5 errors."

If a prefix parameter is given, then termination occurs if the parameter is zero. (Hence ~^ is equivalent to ~#^.) If two parameters are given, termination occurs if they are equal. If three parameters are given, termination occurs if the first is less than or equal to the second and the second is less than or equal to the third. Of course, this is useless if all the prefix parameters are constants; at least one of them should be a # or a V parameter.

If ~^ is used within a ~:{ construct, then it merely terminates the current iteration step (because in the standard case it tests for remaining arguments of the current step only); the next iteration step commences immediately. To terminate the entire iteration process, use ~:^.

X3J13 voted in March 1988 to clarify the behavior of ~:^ as follows. It may be used only if the command it would terminate is ~:{ or ~:@{. The entire iteration process is terminated if and only if the sublist that is supplying the arguments for the current iteration step is the last sublist (in the case of terminating a ~:{ command) or the last argument to that call to format (in the case of terminating a ~:@{ command). Note furthermore that while ~^ is equivalent to ~#^ in all circumstances, ~:^ is *not* equivalent to ~:#^ because the latter terminates the entire iteration if and only if no arguments remain for *the current iteration step* (as opposed to no arguments remaining for the entire iteration process).

Here are some examples of the differences in the behaviors of ~^, ~:^, and ~:#^.

```
(format nil
      "~:{/~S~^ ...~}"
      '((hot dog) (hamburger) (ice cream) (french fries)))
⇒ "/HOT .../HAMBURGER/ICE .../FRENCH ..."
```

For each sublist, " ..." appears after the first word unless there are no additional words.

```
(format nil
      "~:{/~S~:^ ...~}"
      '((hot dog) (hamburger) (ice cream) (french fries)))
⇒ "/HOT .../HAMBURGER .../ICE .../FRENCH"
```

For each sublist, " ..." always appears after the first word, unless it is the last sublist, in which case the entire iteration is terminated.

```
(format nil
      "~:{/~S~:#^ ...~}"
      '((hot dog) (hamburger) (ice cream) (french fries)))
⇒ "/HOT .../HAMBURGER"
```

For each sublist, " ..." appears after the first word, but if the sublist has only one word then the entire iteration is terminated.

If `~^` appears within a control string being processed under the control of a `~?` directive, but not within any `~{` or `~<` construct within that string, then the string being processed will be terminated, thereby ending processing of the `~?` directive. Processing then continues within the string containing the `~?` directive at the point following that directive.

If `~^` appears within a `~[` or `~(` construct, then all the commands up to the `~^` are properly selected or case-converted, the `~[` or `~(` processing is terminated, and the outward search continues for a `~{` or `~<` construct to be terminated. For example:

```
(setq tellstr "~@(~@[~R~]~^ ~A.~)")
(format nil tellstr 23) ⇒ "Twenty-three."
(format nil tellstr nil "losers") ⇒ "Losers."
(format nil tellstr 23 "losers") ⇒ "Twenty-three losers."
```

Here are some examples of the use of `~^` within a `~<` construct.

```
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo)
      ⇒ "           FOO"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar)
      ⇒ "FOO         BAR"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar 'baz)
      ⇒ "FOO   BAR   BAZ"
```

X3J13 voted in June 1989 to introduce user-defined directives in the form of the `~/.../` directive. See section 26.4 for details.

The hairiest `format` control string I have ever seen in shown in table 22.13. It started innocently enough as part of the simulator for Connection Machine Lisp [44, 57]; the *xapping* data type, defined by `defstruct`, needed a `:print-function` option so that xappings would print properly. As this data type became more complicated, step by step, so did the `format` control string.

See the description of `set-macro-character` for a discussion of xappings and the `defstruct` definition. Assume that the predicate `xectorp` is true of a xapping if it is a xector, and that the predicate `finite-part-is-xetp` is true if every value in the range is the same as its corresponding index.

Here is a blow-by-blow description of the parts of this format string:

`~:[{~;[~]`	Print "`[`" for a xector, and "`{`" otherwise.
`~:{~S~:[⇒~S~;~*~]~:^ ~}`	Given a list of lists, print the pairs. Each sublist has three elements: the index (or the value if we're printing a xector); a flag that is true for either a xector or xet (in which case no arrow is printed); and the value. Note the use of `~:{` to iterate, and the use of `~:^` to avoid printing a separating space after the final pair (or at all, if there are no pairs).
`~:[~; ~]`	If there were pairs and there are exceptions or an infinite part, print a separating space.
~⟨newline⟩	Do nothing. This merely allows the format control string to be broken across two lines.
`~{~S⇒~^ ~}`	Given a list of exception indices, print them. Note the use of `~{` to iterate, and the use of `~^` to avoid printing a separating space after the final exception (or at all, if there are no exceptions).
`~:[~; ~]`	If there were exceptions and there is an infinite part, print a separating space.
`~[~*~;⇒~S~;⇒~*~]`	Use `~[` to choose one of three cases for printing the infinite part.
`~:[}~;]~]`	Print "`]`" for a xector, and "`}`" otherwise.

22.4 Querying the User

The following functions provide a convenient and consistent interface for asking questions of the user. Questions are printed and the answers are read using the stream `*query-io*`, which normally is synonymous with `*terminal-io*` but can be rebound to another stream for special applications.

[Function] **y-or-n-p** **&optional** *format-string* **&rest** *arguments*

This predicate is for asking the user a question whose answer is either "yes" or "no." It types out a message (if supplied), reads an answer in some implementation-dependent manner (intended to be short and simple, like reading a single character such as `Y` or `N`), and is true if the answer was "yes" or false if the answer was "no."

If the *format-string* argument is supplied and not `nil`, then a `fresh-line` operation is performed; then a message is printed as if the *format-string* and *arguments* were given to `format`. Otherwise it is assumed that any message has already been printed by other means. If you want a question mark at the end of the message, you must put it there yourself; `y-or-n-p` will not add it. However, the message should not contain an explanatory note such as `(Y or N)`, because the nature of the interface provided for `y-or-n-p` by a given implementation might not involve typing a character on a keyboard; `y-or-n-p` will provide such a note if appropriate.

All input and output are performed using the stream in the global variable `*query-io*`.

Here are some examples of the use of `y-or-n-p`:

```
(y-or-n-p "Produce listing file?")
(y-or-n-p "Cannot connect to network host ~S. Retry?" host)
```

`y-or-n-p` should only be used for questions that the user knows are coming or in situations where the user is known to be waiting for a response of some kind. If the user is unlikely to anticipate the question, or if the consequences of the answer might be grave and irreparable, then `y-or-n-p` should not be used because the user might type ahead and thereby accidentally answer the question. For such questions as "Shall I delete all of your files?" it is better to use `yes-or-no-p`.

[Function] **yes-or-no-p** **&optional** *format-string* **&rest** *arguments*

This predicate, like `y-or-n-p`, is for asking the user a question whose answer is either "yes" or "no." It types out a message (if supplied), attracts the user's attention (for example, by ringing the terminal's bell), and reads a reply in some implementation-dependent manner. It is intended that the reply require the user to take more action than just a single keystroke, such as typing the full word `yes` or `no` followed by a newline.

If the *format-string* argument is supplied and not `nil`, then a `fresh-line` operation is performed; then a message is printed as if the *format-string* and *arguments* were given to `format`. Otherwise it is assumed that any message has already been printed by other means. If you want a question mark at the end of the message, you must put it there yourself; `yes-or-no-p` will not add it. However, the message should not contain an explanatory note such as `(Yes or No)` because the nature of the interface provided for `yes-or-no-p` by a given implementation might not involve typing the reply on a keyboard; `yes-or-no-p` will provide such a note if appropriate.

All input and output are performed using the stream in the global variable `*query-io*`.

To allow the user to answer a yes-or-no question with a single character, use `y-or-n-p`. `yes-or-no-p` should be used for unanticipated or momentous questions; this is why it attracts attention and why it requires a multiple-action sequence to answer it.

Таблица 22.11: Standard Bindings for I/O Control Variables

Variable	Value
package	the `common-lisp-user` package
print-array	t
print-base	10
print-case	:upcase
print-circle	nil
print-escape	t
print-gensym	t
print-length	nil
print-level	nil
print-lines	nil *
print-miser-width	nil *
print-pprint-dispatch	nil *
print-pretty	nil
print-radix	nil
print-readably	t
print-right-margin	nil *
read-base	10
read-default-float-format	single-float
read-eval	t
read-suppress	nil
readtable	the standard readtable

* X3J13 voted in June 1989 to introduce the printer control variables `*print-right-margin*`, `*print-miser-width*`, `*print-lines*`, and `*print-pprint-dispatch*` (see section 26.2) but did not specify the values to which `with-standard-io-syntax` should bind them. I recommend that all four should be bound to `nil`.

Таблица 22.12: Основные связывания для переменных для ввода/вывода

Переменная	Значение
package	пакет `common-lisp-user`
print-array	t
print-base	10
print-case	:upcase
print-circle	nil
print-escape	t
print-gensym	t
print-length	nil
print-level	nil
print-lines	nil *
print-miser-width	nil *
print-pprint-dispatch	nil *
print-pretty	nil
print-radix	nil
print-readably	t
print-right-margin	nil *
read-base	10
read-default-float-format	single-float
read-eval	t
read-suppress	nil
readtable	стандартная таблица с макросимволами

* X3J13 voted in June 1989 to introduce the printer control variables `*print-right-margin*`, `*print-miser-width*`, `*print-lines*`, and `*print-pprint-dispatch*` (see section 26.2) but did not specify the values to which `with-standard-io-syntax` should bind them. I recommend that all four should be bound to `nil`.

Таблица 22.13: Print Function for the Xapping Data Type

```
(defun print-xapping (xapping stream depth)
  (declare (ignore depth))
  (format stream
          ;; Are you ready for this one?
          "~:[{~;[~]~:{~S~:[⇒~S~;~*~]~:^ ~}~:[~; ~]~
           ~{~S⇒~^ ~}~:[~; ~]~[~*~;⇒~S~;⇒~*~]~:[}~;]~]"
          ;; Is that clear?
          (xectorp xapping)
          (do ((vp (xectorp xapping))
               (sp (finite-part-is-xetp xapping))
               (d (xapping-domain xapping) (cdr d))
               (r (xapping-range xapping) (cdr r))
               (z '() (cons (list (if vp (car r) (car d))
                                  (or vp sp)
                                  (car r))
                            z)))
              ((null d) (reverse z)))
          (and (xapping-domain xapping)
               (or (xapping-exceptions xapping)
                   (xapping-infinite xapping)))
          (xapping-exceptions xapping)
          (and (xapping-exceptions xapping)
               (xapping-infinite xapping))
          (ecase (xapping-infinite xapping)
            ((nil) 0)
            (:constant 1)
            (:universal 2))
          (xapping-default xapping)
          (xectorp xapping)))
```

See section 22.1.5 for the `defstruct` definition of the `xapping` data type, whose accessor functions are used in this code.

Глава 23

File System Interface

A frequent use of streams is to communicate with a *file system* to which groups of data (files) can be written and from which files can be retrieved.

Common Lisp defines a standard interface for dealing with such a file system. This interface is designed to be simple and general enough to accommodate the facilities provided by "typical" operating system environments within which Common Lisp is likely to be implemented. The goal is to make Common Lisp programs that perform only simple operations on files reasonably portable.

To this end, Common Lisp assumes that files are named, that given a name one can construct a stream connected to a file of that name, and that the names can be fit into a certain canonical, implementation-independent form called a *pathname*.

Facilities are provided for manipulating pathnames, for creating streams connected to files, and for manipulating the file system through pathnames and streams.

23.1 File Names

Common Lisp programs need to use names to designate files. The main difficulty in dealing with names of files is that different file systems have different naming formats for files. For example, here is a table of several file systems (actually, operating systems that provide file systems) and what equivalent file names might look like for each one:

System	File Name
TOPS-20	`<LISPIO>FORMAT.FASL.13`
TOPS-10	`FORMAT.FAS[1,4]`
ITS	`LISPIO;FORMAT FASL`
MULTICS	`>udd>LispIO>format.fasl`
TENEX	`<LISPIO>FORMAT.FASL;13`
VAX/VMS	`[LISPIO]FORMAT.FAS;13`
UNIX	`/usr/lispio/format.fasl`

It would be impossible for each program that deals with file names to know about each different file name format that exists; a new Common Lisp implementation might use a format different from any of its predecessors. Therefore, Common Lisp provides *two* ways to represent file names: *namestrings*, which are strings in the implementation-dependent form customary for the file system, and *pathnames*, which are special abstract data objects that represent file names in an implementation-independent way. Functions are provided to convert between these two representations, and all manipulations of files can be expressed in machine-independent terms by using pathnames.

In order to allow Common Lisp programs to operate in a network environment that may have more than one kind of file system, the pathname facility allows a file name to specify which file system is to be used. In this context, each file system is called a *host*, in keeping with the usual networking terminology.

Different hosts may use different notations for file names. Common Lisp allows customary notation to be used for each host, but also supports a system of logical pathnames that provides a standard framework for naming files in a portable manner (see section 23.1.5).

23.1.1 Pathnames

All file systems dealt with by Common Lisp are forced into a common framework, in which files are named by a Lisp data object of type `pathname`.

A pathname always has six components, described below. These components are the common interface that allows programs to work the same way with different file systems; the mapping of the pathname components into the concepts peculiar to each file system is taken care of by the Common Lisp implementation.

host The name of the file system on which the file resides.

device Corresponds to the "device" or "file structure" concept in many host file systems: the name of a (logical or physical) device containing files.

directory Corresponds to the "directory" concept in many host file systems: the name of a group of related files (typically those belonging to a single user or project).

name The name of a group of files that can be thought of as the "same" file.

type Corresponds to the "filetype" or "extension" concept in many host file systems; identifies the type of file. Files with the same names but different types are usually related in some specific way, for instance, one being a source file, another the compiled form of that source, and a third the listing of error messages from the compiler.

version Corresponds to the "version number" concept in many host file systems. Typically this is a number that is incremented every time the file is modified.

Note that a pathname is not necessarily the name of a specific file. Rather, it is a specification (possibly only a partial specification) of how to access a file. A pathname need not correspond to any file that actually exists, and more than one pathname can refer to the same file. For example, the pathname with a version of "newest" may refer to the same file as a pathname with the same components except a certain number as the version. Indeed, a pathname with version "newest" may refer to different files as time passes, because the meaning of such a pathname depends on the state of the file system. In file systems with such facilities as "links," multiple file names, logical devices, and so on, two pathnames that look quite different may turn out to address the same file. To access a file given a pathname, one must do a file system operation such as `open`.

Two important operations involving pathnames are *parsing* and *merging*. Parsing is the conversion of a namestring (which might be something supplied interactively by the user when asked to supply the name of a file) into a pathname object. This operation is implementation-dependent, because the format of namestrings is implementation-dependent. Merging takes a pathname with missing components and supplies values for those components from a source of defaults.

Not all of the components of a pathname need to be specified. If a component of a pathname is missing, its value is `nil`. Before the file system

interface can do anything interesting with a file, such as opening the file, all the missing components of a pathname must be filled in (typically from a set of defaults). Pathnames with missing components may be used internally for various purposes; in particular, parsing a namestring that does not specify certain components will result in a pathname with missing components.

X3J13 voted in January 1989 to permit any component of a pathname to have the value `:unspecific`, meaning that the component simply does not exist, for file systems in which such a value makes sense. (For example, a UNIX file system usually does not support version numbers, so the version component of a pathname for a UNIX host might be `:unspecific`. Similarly, the file type is usually regarded in a UNIX file system as the part of a name after a period, but some file names contain no periods and therefore have no file types.)

When a pathname is converted to a namestring, the values `nil` and `:unspecific` have the same effect: they are treated as if the component were empty (that is, they each cause the component not to appear in the namestring). When merging, however, only a `nil` value for a component will be replaced with the default for that component; the value `:unspecific` will be left alone as if the field were filled.

The results are undefined if `:unspecific` is supplied to a file system in a component for which `:unspecific` does not make sense for that file system.

Programming hint: portable programs should be prepared to handle the value `:unspecific` in the device, directory, type, or version field in some implementations. Portable programs should not explicitly place `:unspecific` in any field because it might not be permitted in some situations, but portable programs may sometimes do so implicitly (by copying such a value from another pathname, for example).

What values are allowed for components of a pathname depends, in general, on the pathname's host. However, in order for pathnames to be usable in a system-independent way, certain global conventions are adhered to. These conventions are stronger for the type and version than for the other components, since the type and version are explicitly manipulated by many programs, while the other components are usually treated as something supplied by the user that just needs to be remembered and copied from place to place.

The type is always a string or `nil` or `:wild`. It is expected that most programs that deal with files will supply a default type for each file.

The version is either a positive integer or a special symbol. The meanings

of `nil` and `:wild` have been explained above. The keyword `:newest` refers to the largest version number that already exists in the file system when reading a file, or to a version number greater than any already existing in the file system when writing a new file. Some Common Lisp implementors may choose to define other special version symbols. Some semi-standard names, suggested but not required to be supported by every Common Lisp implementation, are `:oldest`, to refer to the smallest version number that exists in the file system; `:previous`, to refer to the version previous to the newest version; and `:installed`, to refer to a version that is officially installed for users (as opposed to a working or development version). Some Common Lisp implementors may also choose to attach a meaning to non-positive version numbers (a typical convention is that `0` is synonymous with `:newest` and `-1` with `:previous`), but such interpretations are implementation-dependent.

The host may be a string, indicating a file system, or a list of strings, of which the first names the file system and the rest may be used for such a purpose as inter-network routing.

X3J13 voted in June 1989 to approve the following clarifications and specifications of precisely what are valid values for the various components of a pathname.

Pathname component value strings never contain the punctuation characters that are used to separate fields in a namestring (for example, slashes and periods as used in UNIX file systems). Punctuation characters appear only in namestrings. Characters used as punctuation can appear in pathname component values with a non-punctuation meaning if the file system allows it (for example, UNIX file systems allow a file name to begin with a period).

When examining pathname components, conforming programs must be prepared to encounter any of the following siutations:

- Any component can be `nil`, which means the component has not been specified.
- Any component can be `:unspecific`, which means the component has no meaning in this particular pathname.
- The device, directory, name, and type can be strings.
- The host can be any object, at the discretion of the implementation.
- The directory can be a list of strings and symbols as described in section 23.1.3.

- The version can be any symbol or any integer. The symbol `:newest` refers to the largest version number that already exists in the file system when reading, overwriting, appending, superseding, or directory-listing an existing file; it refers to the smallest version number greater than any existing version number when creating a new file. Other symbols and integers have implementation-defined meaning. It is suggested, but not required, that implementations use positive integers starting at 1 as version numbers, recognize the symbol `:oldest` to designate the smallest existing version number, and use keyword symbols for other special versions.

When examining wildcard components of a wildcard pathname, conforming programs must be prepared to encounter any of the following additional values in any component or any element of a list that is the directory component:

- The symbol `:wild`, which matches anything.
- A string containing implementation-dependent special wildcard characters.
- Any object, representing an implementation-dependent wildcard pattern.

When constructing a pathname from components, conforming programs must follow these rules:

- Any component may be `nil`. Specifying `nil` for the host may, in some implementations, result in using a default host rather than an actual `nil` value.
- The host, device, directory, name, and type may be strings. There are implementation-dependent limits on the number and type of characters in these strings. A plausible assumption is that letters (of a single case) and digits are acceptable to most file systems.
- The directory may be a list of strings and symbols as described in section 23.1.3. There are implementation-dependent limits on the length and contents of the list.
- The version may be `:newest`.

- Any component may be taken from the corresponding component of another pathname. When the two pathnames are for different file systems (in implementations that support multiple file systems), an appropriate translation occurs. If no meaningful translation is possible, an error is signaled. The definitions of "appropriate" and "meaningful" are implementation-dependent.

- When constructing a wildcard pathname, the name, type, or version may be `:wild`, which matches anything.

- An implementation might support other values for some components, but a portable program should not use those values. A conforming program can use implementation-dependent values but this can make it non-portable; for example, it might work only with UNIX file systems.

The best way to compare two pathnames for equality is with `equal`, not `eql`. (On pathnames, `eql` is simply the same as `eq`.) Two pathname objects are `equal` if and only if all the corresponding components (host, device, and so on) are equivalent. (Whether or not uppercase and lowercase letters are considered equivalent in strings appearing in components depends on the file name conventions of the file system.) Pathnames that are `equal` should be functionally equivalent.

23.1.2 Case Conventions

Issues of alphabetic case in pathnames are a major source of problems. In some file systems, the customary case is lowercase, in some uppercase, in some mixed. Some file systems are case-sensitive (that is, they treat `FOO` and `foo` as different file names) and others are not.

There are two kinds of pathname case portability problems: moving programs from one Common Lisp to another, and moving pathname component values from one file system to another. The solution to the first problem is the requirement that all Common Lisp implementations that support a particular file system must use compatible representations for pathname component values. The solution to the second problem is the use of a common representation for the least-common-denominator pathname component values that exist on all interesting file systems.

Requiring a common representation directly conflicts with the desire among programmers that use only one file system to work with the local

conventions and to ignore issues of porting to other file systems. The common representation cannot be the same as local (varying) conventions.

X3J13 voted in June 1989 to add a keyword argument `:case` to each of the functions `make-pathname`, `pathname-host`, `pathname-device`, `pathname-directory`, `pathname-name`, and `pathname-type`. The possible values for the argument are `:common` and `:local`. The default is `:local`.

The value `:local` means that strings given to `make-pathname` or returned by any of the pathname component accessors follow the local file system's conventions for alphabetic case. Strings given to `make-pathname` will be used exactly as written if the file system supports both cases. If the file system supports only one case, the strings will be translated to that case.

The value `:common` means that strings given to `make-pathname` or returned by any of the pathname component accessors follow this common convention:

- All uppercase means that a file system's customary case will be used.
- All lowercase means that the opposite of the customary case will be used.
- Mixed case represents itself.

Uppercase is used as the common case for no better reason than consistency with Lisp symbols. The second and third points allow translation from local representation to common and back to be information-preserving. (Note that translation from common to local representation and back may or may not be information-preserving, depending on the nature of the local representation.)

Namestrings always use `:local` file system case conventions.

Finally, `merge-pathnames` and `translate-pathname` map customary case in the input pathnames into customary case in the output pathname.

Examples of possible use of this convention:

- TOPS-20 is case-sensitive and prefers uppercase, translating lowercase to uppercase unless escaped with `^V`; for a TOPS-20–based file system, a Common Lisp implementation should use identical representations for common and local.
- UNIX is case-sensitive and prefers lowercase; for a UNIX-based file system, a Common Lisp implementation should translate between common and local representations by inverting the case of non-mixed-case strings.

- VAX/VMS is uppercase-only (that is, the file system translates all file name arguments to uppercase); for a VAX/VMS-based file system, a Common Lisp implementation should translate common representation to local by converting to uppercase and should translate local representation to common with no change.

- The Macintosh operating system is case-insensitive and prefers lowercase, but remembers the cases of letters actually used to name a file; for a Macintosh-based file system, a Common Lisp implementation should translate between common and local representations by inverting the case of non-mixed-case strings and should ignore case when determining whether two pathnames are `equal`.

Here are some examples of this behavior. Assume that the host `T` runs TOPS-20, `U` runs UNIX, `V` runs VAX/VMS, and `M` runs the Macintosh operating system.

```
;;; Returns two values: the PATHNAME-NAME from a namestring
;;; in :COMMON and :LOCAL representations (in that order).
(defun pathname-example (name)
  (let ((path (parse-namestring name))))
    (values (pathname-name path :case :common)
            (pathname-name path :case :local))))

                                        ;Common     Local
(pathname-example "T:<ME>FOO.LISP")       ⇒ "FOO" and "FOO"
(pathname-example "T:<ME>foo.LISP")       ⇒ "FOO" and "FOO"
(pathname-example "T:<ME>^Vf^Vo^Vo.LISP") ⇒ "foo" and "foo"
(pathname-example "T:<ME>TeX.LISP")       ⇒ "TEX" and "TEX"
(pathname-example "T:<ME>T^VeX.LISP")     ⇒ "TeX" and "TeX"
(pathname-example "U:/me/FOO.lisp")       ⇒ "foo" and "FOO"
(pathname-example "U:/me/foo.lisp")       ⇒ "FOO" and "foo"
(pathname-example "U:/me/TeX.lisp")       ⇒ "TeX" and "TeX"
(pathname-example "V:[me]FOO.LISP")       ⇒ "FOO" and "FOO"
(pathname-example "V:[me]foo.LISP")       ⇒ "FOO" and "FOO"
(pathname-example "V:[me]TeX.LISP")       ⇒ "TEX" and "TEX"
(pathname-example "M:FOO.LISP")           ⇒ "foo" and "FOO"
(pathname-example "M:foo.LISP")           ⇒ "FOO" and "foo"
(pathname-example "M:TeX.LISP")           ⇒ "TeX" and "TeX"
```

The following example illustrates the creation of new pathnames. The name is converted from common representation to local because namestrings always use local conventions.

```
(defun make-pathname-example (h n)
  (namestring (make-pathname :host h :name n :case :common))

(make-pathname-example "T" "FOO") ⇒ "T:FOO"
(make-pathname-example "T" "foo") ⇒ "T:^Vf^Vo^Vo"
(make-pathname-example "T" "TeX") ⇒ "T:T^VeX"
(make-pathname-example "U" "FOO") ⇒ "U:foo"
```

(make-pathname-example "U" "foo") ⇒ "U:FOO"
(make-pathname-example "U" "TeX") ⇒ "U:TeX"
(make-pathname-example "V" "FOO") ⇒ "V:FOO"
(make-pathname-example "V" "foo") ⇒ "V:FOO"
(make-pathname-example "V" "TeX") ⇒ "V:TeX"
(make-pathname-example "M" "FOO") ⇒ "M:foo"
(make-pathname-example "M" "foo") ⇒ "M:FOO"
(make-pathname-example "M" "TeX") ⇒ "M:TeX"

A big advantage of this set of conventions is that one can, for example, call `make-pathname` with `:type "LISP"` and `:case :common`, and the result will appear in a namestring as `.LISP` or `.lisp`, whichever is appropriate.

23.1.3 Structured Directories

X3J13 voted in June 1989 to define a specific pathname component format for structured directories.

The value of a pathname's directory component may be a list. The *car* of the list should be a keyword, either `:absolute` or `:relative`. Each remaining element of the list should be a string or a symbol (see below). Each string names a single level of directory structure and should consist of only the directory name without any punctuation characters.

A list whose *car* is the symbol `:absolute` represents a directory path starting from the root directory. For example, the list `(:absolute)` represents the root directory itself; the list `(:absolute "foo" "bar" "baz")` represents the directory that in a UNIX file system would be called `/foo/bar/baz`.

A list whose *car* is the symbol `:relative` represents a directory path starting from a default directory. The list `(:relative)` has the same meaning as `nil` and hence normally is not used. The list `(:relative "foo" "bar")` represents the directory named `bar` in the directory named `foo` in the default directory.

In place of a string, at any point in the list, a symbol may occur to indicate a special file notation. The following symbols have standard meanings.

`:wild` Wildcard match of one level of directory structure

`:wild-inferiors` Wildcard match of any number of directory levels

`:up` Go upward in directory structure (semantic)

`:back` Go upward in directory structure (syntactic)

(See section 23.1.4 for a discussion of wildcard pathnames.)

Implementations are permitted to add additional objects of any non-string type if necessary to represent features of their file systems that cannot be represented with the standard strings and symbols. Supplying any non-string, including any of the symbols listed below, to a file system for which it does not make sense signals an error of type `file-error`. For example, most implementations of the UNIX file system do not support `:wild-inferiors`. Any directory list in which `:absolute` or `:wild-inferiors` is immediately followed by `:up` or `:back` is illegal and when processed causes an error to be signaled.

The keyword `:back` has a "syntactic" meaning that depends only on the pathname and not on the contents of the file system. The keyword `:up` has a "semantic" meaning that depends on the contents of the file system; to resolve a pathname containing `:up` to a pathname whose directory component contains only `:absolute` and strings requires a search of the file system. Note that use of `:up` instead of `:back` can result in designating a different actual directory only in file systems that support multiple names for directories, perhaps via symbolic links. For example, suppose that there is a directory link such that

(:absolute "X" "Y") is linked to (:absolute "A" "B")

and there also exist directories

(:absolute "A" "Q") and (:absolute "X" "Q")

Then

(:absolute "X" "Y" :up "Q") designates (:absolute "A" "Q")

but

(:absolute "X" "Y" :back "Q") designates (:absolute "X" "Q")

If a string is used as the value of the `:directory` argument to `make-pathname`, it should be the name of a top-level directory and should not contain any punctuation characters. Specifying a string *s* is equivalent to specifying the list `(:absolute` *s*`)`. Specifying the symbol `:wild` is equivalent to specifying the list `(:absolute :wild-inferiors)` (or `(:absolute :wild)` in a file system that does not support `:wild-inferiors`).

The function `pathname-directory` always returns `nil`, `:unspecific`, or a list—never a string, never `:wild`. If a list is returned, it is not guaranteed to be freshly consed; the consequences of modifying this list are undefined.

In non-hierarchical file systems, the only valid list values for the directory component of a pathname are `(:absolute` *s*`)` (where *s* is a string) and `(:absolute :wild)`. The keywords `:relative`, `:wild-inferiors`, `:up`, and `:back` are not used in non-hierarchical file systems.

Pathname merging treats a relative directory specially. Let *pathname* and *defaults* be the first two arguments to `merge-pathnames`. If `(pathname-directory` *pathname*`)` is a list whose *car* is `:relative`, and `(pathname-directory` *defaults*`)` is a list, then the merged directory is the value of

```
(append (pathname-directory defaults)
        (cdr    ;Remove :relative from the front
          (pathname-directory pathname)))
```

except that if the resulting list contains a string or `:wild` immediately followed by `:back`, both of them are removed. This removal of redundant occurrences of `:back` is repeated as many times as possible. If `(pathname-directory` *defaults*`)` is not a list or `(pathname-directory` *pathname*`)` is not a list whose *car* is `:relative`, the merged directory is the value of

```
(or (pathname-directory pathname)
    (pathname-directory defaults))
```

A relative directory in the pathname argument to a function such as `open` is merged with the value of `*default-pathname-defaults*` before the file system is accessed.

Here are some examples of the use of structured directories. Suppose that host `L` supports a Symbolics Lisp Machine file system, host `U` supports a UNIX file system, and host `V` supports a VAX/VMS file system.

```
(pathname-directory (parse-namestring "V:[FOO.BAR]BAZ.LSP"))
  ⇒ (:ABSOLUTE "FOO" "BAR")

(pathname-directory (parse-namestring "U:/foo/bar/baz.lisp"))
  ⇒ (:ABSOLUTE "foo" "bar")

(pathname-directory (parse-namestring "U:../baz.lisp"))
  ⇒ (:RELATIVE :UP)

(pathname-directory (parse-namestring "U:/foo/bar/../mum/baz"))
  ⇒ (:ABSOLUTE "foo" "bar" :UP "mum")

(pathname-directory (parse-namestring "U:bar/../../ztesch/zip"))
  ⇒ (:RELATIVE "bar" :UP :UP "ztesch")

(pathname-directory (parse-namestring "L:>foo>**>bar>baz.lisp"))
  ⇒ (:ABSOLUTE "FOO" :WILD-INFERIORS "BAR")
```

```
(pathname-directory (parse-namestring "L:>foo>*>bar>baz.lisp"))
  ⇒ (:ABSOLUTE "FOO" :WILD "BAR")
```

23.1.4 Extended Wildcards

Some file systems provide more complex conventions for wildcards than simple component-wise wildcards representable by `:wild`. For example, the namestring `"F*O"` might mean a normal three-character name; a three-character name with the middle character wild; a name with at least two characters, beginning with `F` and ending with `O`; or perhaps a wild match spanning multiple directories. Similarly, the namestring `">foo>**>bar>"` might imply that the middle directory is named `"**"`; the middle directory is `:wild`; there are zero or more middle directories that are `:wild`; or perhaps that the middle directory name matches any two-letter name. Some file systems support even more complex wildcards, such as regular expressions.

X3J13 voted in June 1989 to provide some facilities for dealing with more general wildcard pathnames in a fairly portable manner.

[Function] **wild-pathname-p** *pathname* **&optional** *field-key*

Tests a pathname for the presence of wildcard components. If the first argument is not a pathname, string, or file stream, an error of type `type-error` is signaled.

If no *field-key* is provided, or the *field-key* is `nil`, the result is true if and only if *pathname* has any wildcard components.

If a non-null *field-key* is provided, it must be one of `:host`, `:device`, `:directory`, `:name`, `:type`, or `:version`. In this case, the result is true if and only if the indicated component of *pathname* is a wildcard.

Note that X3J13 voted in June 1989 to specify that an implementation need not support wildcards in all fields; the only requirement is that the name, type, or version may be `:wild`. However, portable programs should be prepared to encounter either `:wild` or implementation-dependent wildcards in any pathname component. The function `wild-pathname-p` provides a portable way for testing the presence of wildcards.

[Function] **pathname-match-p** *pathname wildname*

This predicate is true if and only if the *pathname* matches the *wildname*. The matching rules are implementation-defined but should be consistent with the behavior of the `directory` function. Missing components of *wildname* default to `:wild`.

If either argument is not a pathname, string, or file stream, an error of type `type-error` is signaled. It is valid for *pathname* to be a wild pathname; a wildcard field in *pathname* will match only a wildcard field in *wildname*; that is, `pathname-match-p` is not commutative. It is valid for *wildname* to be a non-wild pathname; I believe that in this case `pathname-match-p` will have the same behavior as `equal`, though the X3J13 specification did not say so.

[Function] **translate-pathname** *source from-wildname to-wildname* **&key**

Translates the pathname *source*, which must match *from-wildname*, into a corresponding pathname (call it *result*), which is constructed so as to match *to-wildname*, and returns *result*.

The pathname *result* is a copy of *to-wildname* with each missing or wildcard field replaced by a portion of *source*; for this purpose a wildcard field is a pathname component with a value of `:wild`, a `:wild` element of a list-valued directory component, or an implementation-defined portion of a component, such as the `*` in the complex wildcard string `"foo*bar"` that some implementations support. An implementation that adds other wildcard features, such as regular expressions, must define how `translate-pathname` extends to those features. A missing field is a pathname component that is `nil`.

The portion of *source* that is copied into *result* is implementation-defined. Typically it is determined by the user interface conventions of the file systems involved. Usually it is the portion of *source* that matches a wildcard field of *from-wildname* that is in the same position as the missing or wildcard field of *to-wildname*. If there is no wildcard field in *from-wildname* at that position, then usually it is the entire corresponding pathname component of *source* or, in the case of a list-valued directory component, the entire corresponding list element. For example, if the name components of *source*, *from-wildname*, and *to-wildname* are `"gazonk"`, `"gaz*"`, and `"h*"` respectively, then in most file systems the wildcard fields of the name component of *from-wildname* and *to-wildname* are each `"*"`, the matching portion of *source* is `"onk"`,

and the name component of *result* is `"honk"`; however, the exact behavior of `translate-pathname` is not dictated by the Common Lisp language and may vary according to the user interface conventions of the file systems involved.

During the copying of a portion of *source* into *result*, additional implementation-defined translations of alphabetic case or file naming conventions may occur, especially when *from-wildname* and *to-wildname* are for different hosts.

If any of the first three arguments is not a pathname, string, or file stream, an error of type `type-error` is signaled. It is valid for *source* to be a wild pathname; in general this will produce a wild *result* pathname. It is valid for *from-wildname* or *to-wildname* or both to be non-wild. An error is signaled if the *source* pathname does not match the *from-wildname*, that is, if `(pathname-match-p` *source from-wildname*`)` would not be true.

There are no specified keyword arguments for `translate-pathname`, but implementations are permitted to extend it by adding keyword arguments. There is one specified return value from `translate-pathname`; implementations are permitted to extend it by returning additional values.

Here is an implementation suggestion. One file system performs this operation by examining corresponding pieces of the three pathnames in turn, where a piece is a pathname component or a list element of a structured component such as a hierarchical directory. Hierarchical directory elements in *from-wildname* and *to-wildname* are matched by whether they are wildcards, not by depth in the directory hierarchy. If the piece in *to-wildname* is present and not wild, it is copied into the result. If the piece in *to-wildname* is `:wild` or `nil`, the corresponding piece in *source* is copied into the result. Otherwise, the piece in *to-wildname* might be a complex wildcard such as `"foo*bar"`; the portion of the piece in *source* that matches the wildcard portion of the corresponding piece in *from-wildname* (or the entire *source* piece, if the *from-wildname* piece is not wild and therefore equals the *source* piece) replaces the wildcard portion of the piece in *to-wildname* and the value produced is used in the result.

X3J13 voted in June 1989 to require `translate-pathname` to map customary case in argument pathnames to the customary case in returned pathnames (see section 23.1.2).

Here are some examples of the use of the new wildcard pathname facilities. These examples are not portable. They are written to run with particular file systems and particular wildcard conventions and are intended to be illustrative, not prescriptive. Other implementations may behave differently.

```
(wild-pathname-p (make-pathname :name :wild)) ⇒ t
(wild-pathname-p (make-pathname :name :wild) :name) ⇒ t
(wild-pathname-p (make-pathname :name :wild) :type) ⇒ nil
(wild-pathname-p (pathname "S:>foo>**>")) ⇒ t          ;Maybe
(wild-pathname-p (make-pathname :name "F*O")) ⇒ t      ;Probably
```

One cannot rely on `rename-file` to handle wild pathnames in a predictable manner. However, one can use `translate-pathname` explicitly to control the process.

```
(defun rename-files (from to)
  "Rename all files that match the first argument by
   translating their names to the form of the second
   argument. Both arguments may be wild pathnames."
  (dolist (file (directory from))
    ;; DIRECTORY produces only pathnames that match from-wildname.
    (rename-file file (translate-pathname file from to))))
```

Assuming one particular set of popular wildcard conventions, this function might exhibit the following behavior. Not all file systems will run this example exactly as written.

```
(rename-files "/usr/me/*.lisp" "/dev/her/*.l")
  renames  /usr/me/init.lisp
to  /dev/her/init.l

(rename-files "/usr/me/pcl*/*" "/sys/pcl/*/")
  renames  /usr/me/pcl-5-may/low.lisp
to  /sys/pcl/pcl-5-may/low.lisp
  (in some file systems the result might be /sys/pcl/5-may/low.lisp)

(rename-files "/usr/me/pcl*/*" "/sys/library/*/")
  renames  /usr/me/pcl-5-may/low.lisp
to  /sys/library/pcl-5-may/low.lisp
  (in some file systems the result might be /sys/library/5-may/low.lisp)
```

```
(rename-files "/usr/me/foo.bar" "/usr/me2/")
  renames  /usr/me/foo.bar
to  /usr/me2/foo.bar

(rename-files "/usr/joe/*-recipes.text"
              "/usr/jim/personal/cookbook/joe's-*-rec.text")
  renames  /usr/joe/lamb-recipes.text
to  /usr/jim/personal/cookbook/joe's-lamb-rec.text
  renames  /usr/joe/veg-recipes.text
to  /usr/jim/personal/cookbook/joe's-veg-rec.text
  renames  /usr/joe/cajun-recipes.text
to  /usr/jim/personal/cookbook/joe's-cajun-rec.text
  renames  /usr/joe/szechuan-recipes.text
to  /usr/jim/personal/cookbook/joe's-szechuan-rec.text
```

The following examples use UNIX syntax and the wildcard conventions of one particular version of UNIX.

```
(namestring
  (translate-pathname "/usr/dmr/hacks/frob.l"
                      "/usr/d*/hacks/*.l"
                      "/usr/d*/backup/hacks/backup-*.*"))
  ⇒ "/usr/dmr/backup/hacks/backup-frob.l"
```

```
(namestring
  (translate-pathname "/usr/dmr/hacks/frob.l"
                      "/usr/d*/hacks/fr*.l"
                      "/usr/d*/backup/hacks/backup-*.*"))
  ⇒ "/usr/dmr/backup/hacks/backup-ob.l"
```

The following examples are similar to the preceding examples but use two different hosts; host `U` supports a UNIX file system and host `V` supports a VAX/VMS file system. Note the translation of file type (from `l` to `LSP`) and the change of alphabetic case conventions.

```
(namestring
  (translate-pathname "U:/usr/dmr/hacks/frob.l"
                      "U:/usr/d*/hacks/*.l"
                      "V:SYS$DISK:[D*.BACKUP.HACKS]BACKUP-*.*"))
  ⇒ "V:SYS$DISK:[DMR.BACKUP.HACKS]BACKUP-FROB.LSP"
```

```
(namestring
  (translate-pathname "U:/usr/dmr/hacks/frob.l"
                      "U:/usr/d*/hacks/fr*.l"
                      "V:SYS$DISK:[D*.BACKUP.HACKS]BACKUP-*.*"))
  ⇒ "V:SYS$DISK:[DMR.BACKUP.HACKS]BACKUP-OB.LSP"
```

The next example is a version of the function `translate-logical-pathname` (simplified a bit) for a logical host named `FOO`. The points of interest are the use of `pathname-match-p` as a `:test` argument for `assoc` and the use of `translate-pathname` as a substrate for `translate-logical-pathname`.

```
(define-condition logical-translation-error (file-error))

(defun my-translate-logical-pathname (pathname &key rules)
  (let ((rule (assoc pathname rules :test #'pathname-match-p)))
    (unless rule
      (error 'logical-translation-error :pathname pathname))
    (translate-pathname pathname (first rule) (second rule))))

(my-translate-logical-pathname
  "FOO:CODE;BASIC.LISP"
  :rules '(("FOO:DOCUMENTATION;" "U:/doc/foo/")
           ("FOO:CODE;"          "U:/lib/foo/")
           ("FOO:PATCHES;*;"     "U:/lib/foo/patch/*/")))
  ⇒ #P"U:/lib/foo/basic.l"
```

23.1.5 Logical Pathnames

Pathname values are not portable, but sometimes they must be mentioned in a program (for example, the names of files containing the program and the data used by the program).

X3J13 voted in June 1989 to provide some facilities for portable pathname values. The idea is to provide a portable framework for pathname values; these logical pathnames are then mapped to physical (that is, actual) pathnames by a set of implementation-dependent or site-dependent rules. The logical pathname facility therefore separates the concerns of program writing and user software architecture from the details of how a software system is embedded in a particular file system or operating environment.

Pathname values are not portable because not all Common Lisp implementations use the same operating system and file name syntax varies widely among operating systems. In addition, corresponding files at two different sites may have different names even when the operating system is the same; for example, they may be on different directories or different devices. The Common Lisp logical pathname system defines a particular pathname structure and namestring syntax that must be supported by all implementations.

[Class] **logical-pathname**

This is a subclass of `pathname`.

Syntax of Logical Pathname Namestrings

The syntax of a logical pathname namestring is as follows:

logical-namestring ::= [*host* **:**] [**;**] {*directory* **;**}* [*name*] [**.** *type* [**.** *version*]]

Note that a logical namestring has no *device* portion.

host ::= *word*
directory ::= *word* | *wildcard-word* | *wildcard-inferiors*
name ::= *word* | *wildcard-word*
type ::= *word* | *wildcard-word*
version ::= *word* | *wildcard-word*
word ::= {*letter* | *digit* | **-**}+
wildcard-word ::= [*word*] ***** {*word* *****}* [*word*]
wildcard-inferiors ::= ******

A *word* consists of one or more uppercase letters, digits, and hyphens.

A *wildcard word* consists of one or more asterisks, uppercase letters, digits, and hyphens, including at least one asterisk, with no two asterisks adjacent. Each asterisk matches a sequence of zero or more characters. The wildcard word `*` parses as `:wild`; all others parse as strings.

Lowercase letters may also appear in a word or wildcard word occurring in a namestring. Such letters are converted to uppercase when the namestring is converted to a pathname. The consequences of using other characters are unspecified.

The *host* is a word that has been defined as a logical pathname host by using `setf` with the function `logical-pathname-translations`.

There is no device, so the device component of a logical pathname is always `:unspecific`. No other component of a logical pathname can be `:unspecific`.

Each *directory* is a word, a wildcard word, or `**` (which is parsed as `:wild-inferiors`). If a semicolon precedes the directories, the directory component is relative; otherwise it is absolute.

The *name* is a word or a wildcard word.

The *type* is a word or a wildcard word.

The *version* is a positive decimal integer or the word `NEWEST` (which is parsed as `:newest`) or `*` (which is parsed as `:wild`). The letters in `NEWEST` can be in either alphabetic case.

The consequences of using any value not specified here as a logical pathname component are unspecified. The null string `""` is not a valid value for any component of a logical pathname, since the null string is not a word or a wildcard word.

Parsing of Logical Pathname Namestrings

Logical pathname namestrings are recognized by the functions `logical-pathname` and `translate-logical-pathname`. The host portion of the logical pathname namestring and its following colon must appear in the namestring arguments to these functions.

The function `parse-namestring` recognizes a logical pathname namestring when the *host* argument is logical or the *defaults* argument is a logical pathname. In this case the host portion of the logical pathname namestring and its following colon are optional. If the host portion of the namestring and the *host* argument are both present and do not match, an

error is signaled. The host argument is logical if it is supplied and came from `pathname-host` of a logical pathname. Whether a host argument is logical if it is a string `equal` to a logical pathname host name is implementation-defined.

The function `merge-pathnames` recognizes a logical pathname namestring when the *defaults* argument is a logical pathname. In this case the host portion of the logical pathname namestring and its following colon are optional.

Whether the other functions that coerce strings to pathnames recognize logical pathname namestrings is implementation-defined. These functions include `parse-namestring` in circumstances other than those described above, `merge-pathnames` in circumstances other than those described above, the `:defaults` argument to `make-pathname`, and the following functions:

`compile-file`	`file-write-date`	`pathname-name`
`compile-file-pathname`	`host-namestring`	`pathname-type`
`delete-file`	`load`	`pathname-version`
`directory`	`namestring`	`probe-file`
`directory-namestring`	`open`	`rename-file`
`dribble`	`pathname`	`translate-pathname`
`ed`	`pathname-device`	`truename`
`enough-namestring`	`pathname-directory`	`wild-pathname-p`
`file-author`	`pathname-host`	`with-open-file`
`file-namestring`	`pathname-match-p`	

Note that many of these functions must accept logical pathnames even though they do not accept logical pathname namestrings.

Using Logical Pathnames

Some real file systems do not have versions. Logical pathname translation to such a file system ignores the version. This implies that a portable program cannot rely on being able to store in a file system more than one version of a file named by a logical pathname.

The type of a logical pathname for a Common Lisp source file is `LISP`. This should be translated into whatever implementation-defined type is appropriate in a physical pathname.

The logical pathname host name `SYS` is reserved for the implementation. The existence and meaning of logical pathnames for logical host `SYS` is implementation-defined.

File manipulation functions must operate with logical pathnames according to the following requirements:

- The following accept logical pathnames and translate them into physical pathnames as if by calling the function `translate-logical-pathname`:

`compile-file`	`ed`	`probe-file`
`compile-file-pathname`	`file-author`	`rename-file`
`delete-file`	`file-write-date`	`truename`
`directory`	`load`	`with-open-file`
`dribble`	`open`	

- Applying the function `pathname` to a stream created by the function `open` or the macro `with-open-file` using a logical pathname produces a logical pathname.

- The functions `truename`, `probe-file`, and `directory` never return logical pathnames.

- Calling `rename-file` with a logical pathname as the second argument returns a logical pathname as the first value.

- `make-pathname` returns a logical pathname if and only if the host is logical. If the `:host` argument to `make-pathname` is supplied, the host is logical if it came from the `pathname-host` of a logical pathname. Whether a `:host` argument is logical if it is a string equal to a logical pathname host name is implementation-defined.

[Function] **logical-pathname** *pathname*

Converts the argument to a logical pathname and returns it. The argument can be a logical pathname, a logical pathname namestring containing a host component, or a stream for which the `pathname` function returns a logical pathname. For any other argument, `logical-pathname` signals an error of type `type-error`.

[Function] **translate-logical-pathname** *pathname* **&key**

Translates a logical pathname to the corresponding physical pathname. The *pathname* argument is first coerced to a pathname. If it is not a pathname, string, or file stream, an error of type `type-error` is signaled.

If the coerced argument is a physical pathname, it is returned.

If the coerced argument is a logical pathname, the first matching translation (according to `pathname-match-p`) of the logical pathname host is applied, as if by calling `translate-pathname`. If the result is a logical pathname, this process is repeated. When the result is finally a physical pathname, it is returned.

If no translation matches a logical pathname, an error of type `file-error` is signaled.

`translate-logical-pathname` may perform additional translations, typically to provide translation of file types to local naming conventions, to accommodate physical file systems with names of limited length, or to deal with special character requirements such as translating hyphens to underscores or uppercase letters to lowercase. Any such additional translations are implementation-defined. Some implementations do no additional translations.

There are no specified keyword arguments for `translate-logical-pathname` but implementations are permitted to extend it by adding keyword arguments. There is one specified return value from `translate-logical-pathname`; implementations are permitted to extend it by returning additional values.

[Function] **logical-pathname-translations** *host*

If the specified *host* is not the host component of a logical pathname and is not a string that has been defined as a logical pathname host name by `setf` of `logical-pathname-translations`, this function signals an error of type `type-error`; otherwise, it returns the list of translations for the specified *host*. Each translation is a list of at least two elements, from-wildname and to-wildname. Any additional elements are implementation-defined. A from-wildname is a logical pathname whose host is the specified *host*. A to-wildname is any pathname. Translations are searched in the order listed, so more specific from-wildnames must precede more general ones.

`(setf (logical-pathname-translations` *host*`)` *translations*`)` sets the list of translations for the logical pathname *host* to *translations*. If *host* is a string that has not previously been used as logical pathname host, a new logical pathname host is defined; otherwise an existing host's translations are replaced. Logical pathname host names are compared with `string-equal`.

When setting the translations list, each from-wildname can be a logical pathname whose host is *host* or a logical pathname namestring *s* parseable by `(parse-namestring` *s host-object*`)`, where *host-object* is an appropriate object for representing the specified *host* to `parse-namestring`. (This circuitous specification dodges the fact that `parse-namestring` does not necessarily accept as its second argument any old string that names a logical host.) Each to-wildname can be anything coercible to a pathname by application of the function `pathname`. If to-wildname coerces to a logical pathname, `translate-logical-pathname` will retranslate the result, repeatedly if necessary.

Implementations may define additional functions that operate on logical pathname hosts (for example, to specify additional translation rules or options).

[Function] **load-logical-pathname-translations** *host*

If a logical pathname host named *host* (a string) is already defined, this function returns `nil`. Otherwise, it searches for a logical pathname host definition in an implementation-defined manner. If none is found, it signals an error. If a definition is found, it installs the definition and returns `t`.

The search used by `load-logical-pathname-translations` should be documented, as logical pathname definitions will be created by users as well as by Lisp implementors. A typical search technique is to look in an implementation-defined directory for a file whose name is derived from the host name in an implementation-defined fashion.

[Function] **compile-file-pathname** *pathname* **&key** *:output-file*

Returns the pathname that `compile-file` would write into, if given the same arguments. If the pathname argument is a logical pathname and the `:output-file` argument is unspecified, the result is a logical pathname. If an implementation supports additional keyword arguments to `compile-file`, `compile-file-pathname` must accept the same arguments.

Examples of the Use of Logical Pathnames

Here is a very simple example of setting up a logical pathname host named `FOO`. Suppose that no translations are necessary to get around file system restrictions, so all that is necessary is to specify the root of the physical

directory tree that contains the logical file system. The namestring syntax in the to-wildname is implementation-specific.

```
(setf (logical-pathname-translations "foo")
      '(("**;*.*.*"           "MY-LISPM:>library>foo>**>")))
```

The following is a sample use of that logical pathname. All return values are of course implementation-specific; all of the examples in this section are of course meant to be illustrative and not prescriptive.

```
(translate-logical-pathname "foo:bar;baz;mum.quux.3")
  ⇒ #P"MY-LISPM:>library>foo>bar>baz>mum.quux.3"
```

Next we have a more complex example, dividing the files among two file servers (`U`, supporting a UNIX file system, and `V`, supporting a VAX/VMS file system) and several different directories. This UNIX file system doesn't support `:wild-inferiors` in the directory, so each directory level must be translated individually. No file name or type translations are required except for `.MAIL` to `.MBX`. The namestring syntax used for the to-wildnames is implementation-specific.

```
(setf (logical-pathname-translations "prog")
      '(("RELEASED;*.*.*"     "U:/sys/bin/my-prog/")
        ("RELEASED;*;*.*.*"   "U:/sys/bin/my-prog/*/")
        ("EXPERIMENTAL;*.*.*"
                              "U:/usr/Joe/development/prog/")
        ("EXPERIMENTAL;DOCUMENTATION;*.*.*"
                              "V:SYS$DISK:[JOE.DOC]")
        ("EXPERIMENTAL;*;*.*.*"
                              "U:/usr/Joe/development/prog/*/")
        ("MAIL;**;*.MAIL"     "V:SYS$DISK:[JOE.MAIL.PROG...]*.MBX")
        ))
```

Here are sample uses of logical host `PROG`. All return values are of course implementation-specific.

```
(translate-logical-pathname "prog:mail;save;ideas.mail.3")
  ⇒ #P"V:SYS$DISK:[JOE.MAIL.PROG.SAVE]IDEAS.MBX.3"

(translate-logical-pathname "prog:experimental;spreadsheet.c")
  ⇒ #P"U:/usr/Joe/development/prog/spreadsheet.c"
```

Suppose now that we have a program that uses three files logically named `MAIN.LISP`, `AUXILIARY.LISP`, and `DOCUMENTATION.LISP`. The following translations might be provided by a software supplier as examples.

For a UNIX file system with long file names:

```
(setf (logical-pathname-translations "prog")
      '(("CODE;*.*.*"         "/lib/prog/")))

(translate-logical-pathname "prog:code;documentation.lisp")
  ⇒ #P"/lib/prog/documentation.lisp"
```

For a UNIX file system with 14-character file names, using `.lisp` as the type:

```
(setf (logical-pathname-translations "prog")
      '(("CODE;DOCUMENTATION.*.*" "/lib/prog/docum.*")
        ("CODE;*.*.*"             "/lib/prog/")))

(translate-logical-pathname "prog:code;documentation.lisp")
  ⇒ #P"/lib/prog/docum.lisp"
```

For a UNIX file system with 14-character file names, using `.l` as the type (the second translation shortens the compiled file type to `.b`):

```
(setf (logical-pathname-translations "prog")
      `(("**;*.LISP.*"      ,(logical-pathname "PROG:**;*.L.*"))
        (,(compile-file-pathname
            (logical-pathname "PROG:**;*.LISP.*"))
                        ,(logical-pathname "PROG:**;*.B.*"))
        ("CODE;DOCUMENTATION.*.*" "/lib/prog/documentatio.*")
        ("CODE;*.*.*"             "/lib/prog/")))
```

```
(translate-logical-pathname "prog:code;documentation.lisp")
  ⇒ #P"/lib/prog/documentatio.l"
```

Discussion of Logical Pathnames

Large programs can be moved between sites without changing any pathnames, provided all pathnames used are logical. A portable system construction tool can be created that operates on programs defined as sets of files named by logical pathnames.

Logical pathname syntax was chosen to be easily translated into the formats of most popular file systems, while still being powerful enough for storing large programs. Although they have hierarchical directories, extended wildcard matching, versions, and no limit on the length of names, logical pathnames can be mapped onto a less capable real file system by translating each directory that is used into a flat directory name, processing wildcards in the Lisp implementation rather than in the file system, treating all versions as `:newest`, and using translations to shorten long names.

Logical pathname words are restricted to non-case-sensitive letters, digits, and hyphens to avoid creating problems with real file systems that support limited character sets for file naming. (If logical pathnames were case-sensitive, it would be very difficult to map them into a file system that is not sensitive to case in its file names.)

It is not a goal of logical pathnames to be able to represent all possible file names. Their goal is rather to represent just enough file names to be useful for storing software. Real pathnames, in contrast, need to provide a uniform interface to all possible file names, including names and naming conventions that are not under the control of Common Lisp.

The choice of logical pathname syntax, using colon, semicolon, and period, was guided by the goals of being visually distinct from real file systems and minimizing the use of special characters.

The `logical-pathname` function is separate from the `pathname` function so that the syntax of logical pathname namestrings does not constrain the syntax of physical pathname namestrings in any way. Logical pathname syntax must be defined by Common Lisp so that logical pathnames can be

conveniently exchanged between implementations, but physical pathname syntax is dictated by the operating environments.

The `compile-file-pathname` function and the specification of `LISP` as the type of a logical pathname for a Common Lisp source file together provide enough information about compilation to make possible a portable system construction tool. Suppose that it is desirable to call `compile-file` only if the source file is newer than the compiled file. For this to succeed, it must be possible to know the name of the compiled file without actually calling `compile-file`. In some implementations the compiler produces one of several file types, depending on a variety of implementation-dependent circumstances, so it is not sufficient simply to prescribe a standard logical file type for compiled files; `compile-file-pathname` provides access to the defaulting that is performed by `compile-file` "in a manner appropriate to the implementation's file system conventions."

The use of the logical pathname host name `SYS` for the implementation is current practice. Standardizing on this name helps users choose logical pathname host names that avoid conflicting with implementation-defined names.

Loading of logical pathname translations from a site-dependent file allows software to be distributed using logical pathnames. The assumed model of software distribution is a division of labor between the supplier of the software and the user installing it. The supplier chooses logical pathnames to name all the files used or created by the software, and supplies examples of logical pathname translations for a few popular file systems. Each example uses an assumed directory and/or device name, assumes local file naming conventions, and provides translations that will translate all the logical pathnames used or generated by the particular software into valid physical pathnames. For a powerful file system these translations can be quite simple. For a more restricted file system, it may be necessary to list an explicit translation for every logical pathname used (for example, when dealing with restrictions on the maximum length of a file name).

The user installing the software decides on which device and directory to store the files and edits the example logical pathname translations accordingly. If necessary, the user also adjusts the translations for local file naming conventions and any other special aspects of the user's local file system policy and local Common Lisp implementation. For example, the files might be divided among several file server hosts to share the load. The process of defining site-customized logical pathname translations is quite easy for a

user of a popular file system for which the software supplier has provided an example. A user of a more unusual file system might have to take more time; the supplier can help by providing a list of all the logical pathnames used or generated by the software.

Once the user has created and executed a suitable `setf` form for setting the `logical-pathname-translations` of the relevant logical host, the software can be loaded and run. It may be necessary to use the translations again, or on another workstation at the same site, so it is best to save the `setf` form in the standard place where it can be found later by `load-logical-pathname-translations`. Often a software supplier will include a program for restoring software from the distribution medium to the file system and a program for loading the software from the file system into a Common Lisp; these programs will start by calling `load-logical-pathname-translations` to make sure that the logical pathname host is defined.

Note that the `setf` of `logical-pathname-translations` form isn't part of the program; it is separate and is written by the user, not by the software supplier. That separation and a uniform convention for doing the separation are the key aspects of logical pathnames. For small programs involving only a handful of files, it doesn't matter much. The real benefits come with large programs with hundreds or thousands of files and more complicated situations such as program-generated file names or porting a program developed on a system with long file names onto a system with a very restrictive limit on the length of file names.

23.1.6 Pathname Functions

These functions are what programs use to parse and default file names that have been typed in or otherwise supplied by the user.

Any argument called *pathname* in this book may actually be a pathname, a string or symbol, or a stream. Any argument called *defaults* may likewise be a pathname, a string or symbol, or a stream.

X3J13 voted in March 1988 to change the language so that a symbol is *never* allowed as a pathname argument. More specifically, the following functions are changed to disallow a symbol as a *pathname* argument:

`pathname`	`pathname-device`	`namestring`
`truename`	`pathname-directory`	`file-namestring`
`parse-namestring`	`pathname-name`	`directory-namestring`
`merge-pathnames`	`pathname-type`	`host-namestring`
`pathname-host`	`pathname-version`	`enough-namestring`

(The function `require` was also changed by this vote but was deleted from the language by a vote in January 1989 .) Furthermore, the vote reaffirmed that the following functions do not accept symbols as *file*, *filename*, or *pathname* arguments:

`open`	`rename-file`	`file-write-date`
`with-open-file`	`delete-file`	`file-author`
`load`	`probe-file`	`directory`
`compile-file`		

In older implementations of Lisp that did not have strings, for example MacLisp, symbols were the only means for specifying pathnames. This was convenient only because the file systems of the time allowed only uppercase letters in file names. Typing `(load 'foo)` caused the function `load` to receive the symbol `FOO` (with uppercase letters because of the way symbols are parsed) and therefore to load the file named `FOO`. Now that many file systems, most notably UNIX, support case-sensitive file names, the use of symbols is less convenient and more error-prone.

X3J13 voted in March 1988 to specify that a stream may be used as a `pathname`, `file`, or `filename` argument only if it was created by use of `open` or `with-open-file`, or if it is a synonym stream whose symbol is bound to a stream that may be used as a pathname.

If such a stream is used as a pathname, it is as if the `pathname` function were applied to the stream and the resulting pathname used in place of the stream. This represents the name used to open the file. This may be, but is not required to be, the actual name of the file.

It is an error to attempt to obtain a pathname from a stream created by any of the following:

`make-two-way-stream`	`make-string-input-stream`
`make-echo-stream`	`make-string-output-stream`
`make-broadcast-stream`	`with-input-from-string`
`make-concatenated-stream`	`with-output-to-string`

In the examples, it is assumed that the host named `CMUC` runs the TOPS-20 operating system, and therefore uses TOPS-20 file system syntax; furthermore, an explicit host name is indicated by following the host name with a double colon. Remember, however, that namestring syntax is implementation-dependent, and this syntax is used here purely for the sake of examples.

[Function] **pathname** *pathname*

The `pathname` function converts its argument to be a pathname. The argument may be a pathname, a string or symbol, or a stream; the result is always a pathname.

X3J13 voted in March 1988 not to permit symbols as pathnames and to specify exactly which streams may be used as pathnames .

X3J13 voted in January 1989 to specify that `pathname` is unaffected by whether its argument, if a stream, is open or closed. X3J13 further commented that because some implementations cannot provide the "true name" of a file until the file is closed, in such an implementation `pathname` might, in principle, return a different (perhaps more specific) file name after the stream is closed. However, such behavior is prohibited; `pathname` must return the same pathname after a stream is closed as it would have while the stream was open. See `truename`.

[Function] **truename** *pathname*

The `truename` function endeavors to discover the "true name" of the file associated with the *pathname* within the file system. If the *pathname* is an open stream already associated with a file in the file system, that file is used. The "true name" is returned as a pathname. An error is signaled if an appropriate file cannot be located within the file system for the given *pathname*.

The `truename` function may be used to account for any file name translations performed by the file system, for example.

For example, suppose that `DOC:` is a TOPS-20 logical device name that is translated by the TOPS-20 file system to be `PS:<DOCUMENTATION>`.

```
(setq file (open "CMUC::DOC:DUMPER.HLP"))
(namestring (pathname file)) ⇒ "CMUC::DOC:DUMPER.HLP"
(namestring (truename file))
```

⇒ "CMUC::PS:<DOCUMENTATION>DUMPER.HLP.13"

X3J13 voted in March 1988 not to permit symbols as pathnames and to specify exactly which streams may be used as pathnames .

X3J13 voted in January 1989 to specify that `truename` may be applied to a stream whether the stream is open or closed. X3J13 further commented that because some implementations cannot provide the "true name" of a file until the file is closed, in principle it would be possible in such an implementation for `truename` to return a different file name after the stream is closed. Such behavior is permitted; in this respect `truename` differs from `pathname`.

X3J13 voted in June 1989 to clarify that `truename` accepts only non-wild pathnames; an error is signaled if `wild-pathname-p` would be true of the *pathname* argument.

X3J13 voted in June 1989 to require `truename` to accept logical pathnames (see section 23.1.5). However, `truename` never returns a logical pathname.

[Function] **parse-namestring** *thing* **&optional** *host defaults* **&key** *:start :end :junk-allowed*

This turns *thing* into a pathname. The *thing* is usually a string (that is, a namestring), but it may be a symbol (in which case the print name is used) or a pathname or stream (in which case no parsing is needed, but an error check may be made for matching hosts).

X3J13 voted in March 1988 not to permit symbols as pathnames and to specify exactly which streams may be used as pathnames . The *thing* argument may not be a symbol.

X3J13 voted in June 1989 to require `parse-namestring` to accept logical pathname namestrings (see section 23.1.5).

This function does *not*, in general, do defaulting of pathname components, even though it has an argument named *defaults*; it only does parsing. The *host* and *defaults* arguments are present because in some implementations it may be that a namestring can only be parsed with reference to a particular file name syntax of several available in the implementation. If *host* is non-`nil`, it must be a host name that could appear in the host component of a pathname, or `nil`; if *host* is `nil` then the host name is extracted from the default pathname in *defaults* and used to determine the syntax convention. The *defaults* argument defaults to the value of `*default-pathname-defaults*`.

For a string (or symbol) argument, `parse-namestring` parses a file name within it in the range delimited by the `:start` and `:end` arguments (which are integer indices into *string*, defaulting to the beginning and end of the string).

See chapter 14 for a discussion of `:start` and `:end` arguments.

If `:junk-allowed` is not `nil`, then the first value returned is the pathname parsed, or `nil` if no syntactically correct pathname was seen.

If `:junk-allowed` is `nil` (the default), then the entire substring is scanned. The returned value is the pathname parsed. An error is signaled if the substring does not consist entirely of the representation of a pathname, possibly surrounded on either side by whitespace characters if that is appropriate to the cultural conventions of the implementation.

In either case, the second value is the index into the string of the delimiter that terminated the parse, or the index beyond the substring if the parse terminated at the end of the substring (as will always be the case if `:junk-allowed` is false).

If *thing* is not a string or symbol, then *start* (which defaults to zero in any case) is always returned as the second value.

Parsing an empty string always succeeds, producing a pathname with all components (except the host) equal to `nil`.

Note that if *host* is specified and not `nil`, and *thing* contains a manifest host name, an error is signaled if the hosts do not match.

If *thing* contains an explicit host name and no explicit device name, then it might be appropriate, depending on the implementation environment, for `parse-namestring` to supply the standard default device for that host as the device component of the resulting pathname.

[Function] **merge-pathnames** *pathname* **&optional** *defaults default-version*

X3J13 voted in March 1988 not to permit symbols as pathnames and to specify exactly which streams may be used as pathnames .

X3J13 voted in June 1989 to require `merge-namestrings` to recognize a logical pathname namestring as its first argument if its second argument is a logical pathname (see section 23.1.5).

X3J13 voted in January 1989 to specify that `merge-pathname` is unaffected by whether the first argument, if a stream, is open or closed. If the first

argument is a stream, `merge-pathname` behaves as if the function `pathname` were applied to the stream and the resulting pathname used instead.

X3J13 voted in June 1989 to require `merge-pathnames` to map customary case in argument pathnames to the customary case in returned pathnames (see section 23.1.2).

defaults defaults to the value of `*default-pathname-defaults*`.

default-version defaults to `:newest`.

Here is an example of the use of `merge-pathnames`:

```
(merge-pathnames "CMUC::FORMAT"
                 "CMUC::PS:<LISPIO>.FASL")
  ⇒ a pathname object that re-expressed as a namestring would be
    "CMUC::PS:<LISPIO>FORMAT.FASL.0"
```

Defaulting of pathname components is done by filling in components taken from another pathname. This is especially useful for cases such as a program that has an input file and an output file, and asks the user for the name of both, letting the unsupplied components of one name default from the other. Unspecified components of the output pathname will come from the input pathname, except that the type should default not to the type of the input but to the appropriate default type for output from this program.

The pathname merging operation takes as input a given pathname, a defaults pathname, and a default version, and returns a new pathname. Basically, the missing components in the given pathname are filled in from the defaults pathname, except that if no version is specified the default version is used. The default version is usually `:newest`; if no version is specified the newest version in existence should be used. The default version can be `nil`, to preserve the information that it was missing in the input pathname.

If the given pathname explicitly specifies a host and does not supply a device, then if the host component of the defaults matches the host component of the given pathname, then the device is taken from the defaults; otherwise the device will be the default file device for that host. Next, if the given pathname does not specify a host, device, directory, name, or type, each such component is copied from the defaults. The merging rules for the version are more complicated and depend on whether the pathname specifies a name. If the pathname doesn't specify a name, then the version, if not provided, will come from the defaults, just like the other components. However, if the pathname does specify a name, then the version is not affected

by the defaults. The reason is that the version "belongs to" some other file name and is unlikely to have anything to do with the new one. Finally, if this process leaves the version missing, the default version is used.

The net effect is that if the user supplies just a name, then the host, device, directory, and type will come from the defaults, but the version will come from the default version argument to the merging operation. If the user supplies nothing, or just a directory, the name, type, and version will come over from the defaults together. If the host's file name syntax provides a way to input a version without a name or type, the user can let the name and type default but supply a version different from the one in the defaults.

X3J13 voted in June 1989 to agree to disagree: `merge-pathname` might or might not perform plausibility checking on its arguments to ensure that the resulting pathname can be converted a valid namestring. User beware: this could cause portability problems.

For example, suppose that host `LOSER` constrains file types to be three characters or fewer but host `CMUC` does not. Then `"LOSER::FORMAT"` is a valid namestring and `"CMUC::PS:<LISPIO>.FASL"` is a valid namestring, but

```
(merge-pathnames "LOSER::FORMAT" "CMUC::PS:<LISPIO>.FASL")
```

might signal an error in some implementations because the hypothetical result would be a pathname equivalent to the namestring `"LOSER::FORMAT.FASL"` which is illegal because the file type `FASL` has more than three characters. In other implementations `merge-pathname` might return a pathname but that pathname might cause `namestring` to signal an error.

[Variable] ***default-pathname-defaults***

This is the default pathname-defaults pathname; if any pathname primitive that needs a set of defaults is not given one, it uses this one. As a general rule, however, each program should have its own pathname defaults rather than using this one.

The following example assumes the use of UNIX syntax and conventions.

```
(make-pathname :host "technodrome"
               :directory '(:absolute "usr" "krang")
               :name "shredder")
 ⇒ #P"technodrome:/usr/krang/shredder"
```

X3J13 voted in June 1989 to add a new keyword argument `:case` to `make-pathname`. The new argument description is therefore as follows:

[Function] **make-pathname** **&key** *:host :device :directory :name :type :version :defaults :case*

See section 23.1.2 for a description of the `:case` argument.

X3J13 voted in June 1989 to agree to disagree: `make-pathname` might or might not check on its arguments to ensure that the resulting pathname can be converted to a valid namestring. If `make-pathname` does not check its arguments and signal an error in problematical cases, `namestring` yet might or might not signal an error when given the resulting pathname. User beware: this could cause portability problems.

[Function] **pathnamep** *object*

This predicate is true if *object* is a pathname, and otherwise is false.

(pathnamep x) ≡ (typep x 'pathname)

X3J13 voted in March 1988 not to permit symbols as pathnames and to specify exactly which streams may be used as pathnames .

X3J13 voted in January 1989 to specify that these operations are unaffected by whether the first argument, if a stream, is open or closed. If the first argument is a stream, each operation behaves as if the function `pathname` were applied to the stream and the resulting pathname used instead.

X3J13 voted in June 1989 to add a keyword argument `:case` to all of the pathname accessor functions except `pathname-version`. The new argument descriptions are therefore as follows:

[Function] **pathname-host** *pathname* **&key** *:case*
[Function] **pathname-device** *pathname* **&key** *:case*
[Function] **pathname-directory** *pathname* **&key** *:case*
[Function] **pathname-name** *pathname* **&key** *:case*
[Function] **pathname-type** *pathname* **&key** *:case*
[Function] **pathname-version** *pathname*

See section 23.1.2 for a description of the `:case` argument.

X3J13 voted in June 1989 to specify that `pathname-directory` always returns `nil`, `:unspecific`, or a list—never a string, never `:wild` (see section 23.1.3). If a list is returned, it is not guaranteed to be freshly consed; the consequences of modifying this list are undefined.

[Function] **namestring** *pathname*
[Function] **file-namestring** *pathname*
[Function] **directory-namestring** *pathname*
[Function] **host-namestring** *pathname*
[Function] **enough-namestring** *pathname* **&optional** *defaults*

The *pathname* argument may be a pathname, a string or symbol, or a stream that is or was open to a file. The name represented by *pathname* is returned as a namelist in canonical form.

If *pathname* is a stream, the name returned represents the name used to *open* the file, which may not be the *actual* name of the file (see `truename`).

X3J13 voted in March 1988 not to permit symbols as pathnames and to specify exactly which streams may be used as pathnames .

X3J13 voted in January 1989 to specify that these operations are unaffected by whether the first argument, if a stream, is open or closed. If the first argument is a stream, each operation behaves as if the function `pathname` were applied to the stream and the resulting pathname used instead.

`namestring` returns the full form of the *pathname* as a string. `file-namestring` returns a string representing just the *name*, *type*, and *version* components of the *pathname*; the result of `directory-namestring` represents just the *directory-name* portion; and `host-namestring` returns a string for just the *host-name* portion. Note that a valid namestring cannot necessarily be constructed simply by concatenating some of the three shorter strings in some order.

`enough-namestring` takes another argument, *defaults*. It returns an abbreviated namestring that is just sufficient to identify the file named by *pathname* when considered relative to the *defaults* (which defaults to the value of `*default-pathname-defaults*`). That is, it is required that

(merge-pathnames (enough-namestring *pathname defaults*) *defaults*) ≡
(merge-pathnames (parse-namestring *pathname* nil *defaults*) *defaults*)

in all cases; and the result of `enough-namestring` is, roughly speaking, the shortest reasonable string that will still satisfy this criterion. X3J13 voted in June 1989 to agree to disagree: `make-pathname` and `merge-pathnames` might or might not be able to produce pathnames that cannot be converted to valid namestrings. User beware: this could cause portability problems.

[Function] **user-homedir-pathname** **&optional** *host*

Returns a pathname for the user's "home directory" on *host*. The *host* argument defaults in some appropriate implementation-dependent manner. The concept of "home directory" is itself somewhat implementation-dependent, but from the point of view of Common Lisp it is the directory where the user keeps personal files such as initialization files and mail. If it is impossible to determine this information, then `nil` is returned instead of a pathname; however, `user-homedir-pathname` never returns `nil` if the *host* argument is not specified. This function returns a pathname without any name, type, or version component (those components are all `nil`).

23.2 Opening and Closing Files

When a file is *opened*, a stream object is constructed to serve as the file system's ambassador to the Lisp environment; operations on the stream are reflected by operations on the file in the file system. The act of *closing* the file (actually, the stream) ends the association; the transaction with the file system is terminated, and input/output may no longer be performed on the stream. The stream function `close` may be used to close a file; the functions described below may be used to open them. The basic operation is `open`, but `with-open-file` is usually more convenient for most applications.

[Function] **open** *filename* **&key** *:direction :element-type :if-exists :if-does-not-exist :external-format*

X3J13 voted in June 1989 to add to the function `open` a new keyword argument `:external-format`. This argument did not appear in the preceding argument description in the first edition.

This returns a stream that is connected to the file specified by *filename*. The *filename* is the name of the file to be opened; it may be a string, a pathname, or a stream. (If the *filename* is a stream, then it is not closed first or otherwise affected; it is used merely to provide a file name for the opening of a new stream.)

X3J13 voted in January 1989 to specify that the result of `open`, if it is a stream, is always a stream of type `file-stream`.

X3J13 voted in March 1988 to specify exactly which streams may be used as pathnames. See section 23.1.6.

X3J13 voted in January 1989 to specify that `open` is unaffected by whether the first argument, if a stream, is open or closed. If the first argument is a stream, `open` behaves as if the function `pathname` were applied to the stream and the resulting pathname used instead.

X3J13 voted in June 1989 to clarify that `open` accepts only non-wild pathnames; an error is signaled if `wild-pathname-p` would be true of *filename*.

X3J13 voted in June 1989 to require `open` to accept logical pathnames (see section 23.1.5).

The keyword arguments specify what kind of stream to produce and how to handle errors:

`:direction` This argument specifies whether the stream should handle input, output, or both.

- `:input` The result will be an input stream. This is the default.
- `:output` The result will be an output stream.
- `:io` The result will be a bidirectional stream.
- `:probe` The result will be a no-directional stream (in effect, the stream is created and then closed). This is useful for determining whether a file exists without actually setting up a complete stream.

`:element-type` This argument specifies the type of the unit of transaction for the stream. Anything that can be recognized as being a finite subtype of `character` or `integer` is acceptable. In particular, the following types are recognized:

`character` The unit of transaction is any character, not just a string-character. The functions `read-char` and `write-char` (depending on the value of the `:direction` argument) may be used on the stream. This is the default.

`base-char` The unit of transaction is a base character. The functions `read-char` and `write-char` (depending on the value of the `:direction` argument) may be used on the stream.

`(unsigned-byte` *n*`)` The unit of transaction is an unsigned byte (a non-negative integer) of size *n*. The functions `read-byte` and/or `write-byte` may be used on the stream.

`unsigned-byte` The unit of transaction is an unsigned byte (a non-negative integer); the size of the byte is determined by the file system. The functions `read-byte` and/or `write-byte` may be used on the stream.

`(signed-byte` *n*`)` The unit of transaction is a signed byte of size *n*. The functions `read-byte` and/or `write-byte` may be used on the stream.

`signed-byte` The unit of transaction is a signed byte; the size of the byte is determined by the file system. The functions `read-byte` and/or `write-byte` may be used on the stream.

`bit` The unit of transaction is a bit (values `0` and `1`). The functions `read-byte` and/or `write-byte` may be used on the stream.

`(mod` *n*`)` The unit of transaction is a non-negative integer less than *n*. The functions `read-byte` and/or `write-byte` may be used on the stream.

`:default` The unit of transaction is to be determined by the

file system, based on the file it finds. The type can be determined by using the function `stream-element-type`.

`:if-exists` This argument specifies the action to be taken if the `:direction` is `:output` or `:io` and a file of the specified name already exists. If the direction is `:input` or `:probe`, this argument is ignored.

`:error` Signals an error. This is the default when the version component of the *filename* is not `:newest`.

`:new-version` Creates a new file with the same file name but with a larger version number. This is the default when the version component of the *filename* is `:newest`.

`:rename` Renames the existing file to some other name and then creates a new file with the specified name.

`:rename-and-delete` Renames the existing file to some other name and then deletes it (but does not expunge it, on those systems that distinguish deletion from expunging). Then create a new file with the specified name.

`:overwrite` Uses the existing file. Output operations on the stream will destructively modify the file. If the `:direction` is `:io`, the file is opened in a bidirectional mode that allows both reading and writing. The file pointer is initially positioned at the beginning of the file; however, the file is not truncated back to length zero when it is opened. This mode is most useful when the `file-position` function can be used on the stream.

`:append` Uses the existing file. Output operations on the stream will destructively modify the file. The file pointer is initially positioned at the end of the file. If the `:direction` is `:io`, the file is opened in a bidirectional mode that allows both reading and writing.

`:supersede` Supersedes the existing file. If possible, the implementation should arrange not to destroy the old file until the new stream is closed, against the possibility that the stream will be closed in "abort" mode (see `close`). This differs from `:new-version` in that

`:supersede` creates a new file with the same name as the old one, rather than a file name with a higher version number.

`nil` Does not create a file or even a stream, but instead simply returns `nil` to indicate failure.

If the `:direction` is `:output` or `:io` and the value of `:if-exists` is `:new-version`, then the version of the (newly created) file that is opened will be a version greater than that of any other file in the file system whose other pathname components are the same as those of *filename*.

If the `:direction` is `:input` or `:probe` or the value of `:if-exists` is not `:new-version`, *and* the version component of the *filename* is `:newest`, then the file opened is that file already existing in the file system that has a version greater than that of any other file in the file system whose other pathname components are the same as those of *filename*.

Some file systems permit yet other actions to be taken when a file already exists; therefore, some implementations provide implementation-specific `:if-exist` options.

Заметка для реализации: The various file systems in existence today have widely differing capabilities. A given implementation may not be able to support all of these options in exactly the manner stated. An implementation is required to recognize all of these option keywords and to try to do something "reasonable" in the context of the host operating system. Implementors are encouraged to approximate the semantics specified here as closely as possible.

As an example, suppose that a file system does not support distinct file versions and does not distinguish the notions of deletion and expunging (in some file systems file deletion is reversible until an expunge operation is performed). Then `:new-version` might be treated the same as `:rename` or `:supersede`, and `:rename-and-delete` might be treated the same as `:supersede`.

If it is utterly impossible for an implementation to handle some option in a manner close to what is specified here, it may simply signal an error. The opening of files is an area where complete portability is too much to hope for; the intent here is simply to make things as portable as possible by providing specific names for a range of commonly supportable options.

`:if-does-not-exist` This argument specifies the action to be taken if a file of the specified name does not already exist.

> `:error` Signals an error. This is the default if the `:direction` is `:input`, or if the `:if-exists` argument is `:overwrite` or `:append`.
>
> `:create` Creates an empty file with the specified name and then proceeds as if it had already existed (but do not perform any processing directed by the `:if-exists` argument). This is the default if the `:direction` is `:output` or `:io`, and the `:if-exists` argument is anything but `:overwrite` or `:append`.
>
> `nil` Does not create a file or even a stream, but instead simply returns `nil` to indicate failure. This is the default if the `:direction` is `:probe`.

X3J13 voted in June 1989 to add to the function `open` a new keyword argument `:external-format`.

`:external-format` This argument specifies an implementation-recognized scheme for representing characters in files. The default value is `:default` and is implementation-defined but must support the base characters. An error is signaled if the implementation does recognize the specified format.

This argument may be specified if the `:direction` argument is `:input`, `:output`, or `:io`. It is an error to write a character to the resulting stream that cannot be represented by the specified file format. (However, the `#\Newline` character cannot produce such an error; implementations must provide appropriate line division behavior for all character streams.)

See `stream-external-format`.

When the caller is finished with the stream, it should close the file by using the `close` function. The `with-open-file` form does this automatically, and so is preferred for most purposes. `open` should be used only when the control structure of the program necessitates opening and closing of a file in some way more complex than provided by `with-open-file`. It is suggested

that any program that uses `open` directly should use the special operator `unwind-protect` to close the file if an abnormal exit occurs.

[Макрос] **with-open-file** (stream filename {options}*) {declaration}* {form}*

`with-open-file` evaluates the *forms* of the body (an implicit `progn`) with the variable *stream* bound to a stream that reads or writes the file named by the value of *filename*. The *options* are evaluated and are used as keyword arguments to the function `open`.

When control leaves the body, either normally or abnormally (such as by use of `throw`), the file is automatically closed. If a new output file is being written, and control leaves abnormally, the file is aborted and the file system is left, so far as possible, as if the file had never been opened. Because `with-open-file` always closes the file, even when an error exit is taken, it is preferred over `open` for most applications.

filename is the name of the file to be opened; it may be a string, a pathname, or a stream.

X3J13 voted in March 1988 to specify exactly which streams may be used as pathnames. See section 23.1.6.

X3J13 voted in June 1989 to clarify that `with-open-file` accepts only non-wild pathnames; an error is signaled if `wild-pathname-p` would be true of the *filename* argument.

X3J13 voted in June 1989 to require `with-open-file` to accept logical pathnames (see section 23.1.5).

For example:

```
(with-open-file (ifile name
                       :direction :input)
  (with-open-file (ofile (merge-pathname-defaults ifile
                                                  nil
                                                  "out")
                         :direction :output
                         :if-exists :supersede)
    (transduce-file ifile ofile)))
```

X3J13 voted in June 1989 to specify that the variable *stream* is not always bound to a stream; rather it is bound to whatever would be returned by a

call to `open`. For example, if the options include `:if-does-not-exist nil`, *stream* will be bound to `nil` if the file does not exist. In this case the value of *stream* should be tested within the body of the `with-open-file` form before it is used as a stream. For example:

```
(with-open-file (ifile name
                       :direction :input
                       :if-does-not-exist nil)
  ;; Process the file only if it actually exists.
  (when (streamp name)
    (compile-cobol-program ifile)))
```

Заметка для реализации: While `with-open-file` tries to automatically close the stream on exit from the construct, for robustness it is helpful if the garbage collector can detect discarded streams and automatically close them.

23.3 Renaming, Deleting, and Other File Operations

These functions provide a standard interface to operations provided in some form by most file systems. It may be that some implementations of Common Lisp cannot support them all completely.

[Function] **rename-file** *file new-name*

The specified *file* is renamed to *new-name* (which must be a file name). The *file* may be a string, a pathname, or a stream. If it is an open stream associated with a file, then the stream itself and the file associated with it are affected (if the file system permits).

X3J13 voted in March 1988 to specify exactly which streams may be used as pathnames. See section 23.1.6.

`rename-file` returns three values if successful. The first value is the *new-name* with any missing components filled in by performing a `merge-pathnames` operation using *file* as the defaults. The second value is the `truename` of the file before it was renamed. The third value is the `truename` of the file after it was renamed.

If the renaming operation is not successful, an error is signaled.

X3J13 voted in June 1989 to require `rename-file` to accept logical pathnames (see section 23.1.5).

[Function] **delete-file** *file*

The specified *file* is deleted. The *file* may be a string, a pathname, or a stream. If it is an open stream associated with a file, then the stream itself and the file associated with it are affected (if the file system permits), in which case the stream may or may not be closed immediately, and the deletion may be immediate or delayed until the stream is explicitly closed, depending on the requirements of the file system.

X3J13 voted in March 1988 to specify exactly which streams may be used as pathnames. See section 23.1.6.

`delete-file` returns a non-`nil` value if successful. It is left to the discretion of the implementation whether an attempt to delete a non-existent file is considered to be successful. If the deleting operation is not successful, an error is signaled.

X3J13 voted in June 1989 to require `delete-file` to accept logical pathnames (see section 23.1.5).

[Function] **probe-file** *file*

This predicate is false if there is no file named *file*, and otherwise returns a pathname that is the true name of the file (which may be different from *file* because of file links, version numbers, or other artifacts of the file system). Note that if the *file* is an open stream associated with a file, then `probe-file` cannot return `nil` but will produce the true name of the associated file. See `truename` and the `:probe` value for the `:direction` argument to `open`.

X3J13 voted in March 1988 to specify exactly which streams may be used as pathnames. See section 23.1.6.

X3J13 voted in June 1989 to clarify that `probe-file` accepts only non-wild pathnames; an error is signaled if `wild-pathname-p` would be true of the *file* argument.

X3J13 voted in June 1989 to require `probe-file` to accept logical pathnames (see section 23.1.5). However, `probe-file` never returns a logical pathname.

X3J13 voted in January 1989 to specify that `probe-file` is unaffected by whether the first argument, if a stream, is open or closed. If the first argument

is a stream, `probe-file` behaves as if the function `pathname` were applied to the stream and the resulting pathname used instead. However, X3J13 further commented that the treatment of open streams may differ considerably from one implementation to another; for example, in some operating systems open files are written under a temporary or invisible name and later renamed when closed. In general, programmers writing code intended to be portable should be very careful when using `probe-file`.

[Function] **file-write-date** *file*

file can be a file name or a stream that is open to a file. This returns the time at which the file was created or last written as an integer in universal time format (see section 24.3.1), or `nil` if this cannot be determined.

X3J13 voted in March 1988 to specify exactly which streams may be used as pathnames. See section 23.1.6.

X3J13 voted in June 1989 to clarify that `file-write-date` accepts only non-wild pathnames; an error is signaled if `wild-pathname-p` would be true of the *file* argument.

X3J13 voted in June 1989 to require `file-write-date` to accept logical pathnames (see section 23.1.5).

[Function] **file-author** *file*

file can be a file name or a stream that is open to a file. This returns the name of the author of the file as a string, or `nil` if this cannot be determined.

X3J13 voted in March 1988 to specify exactly which streams may be used as pathnames. See section 23.1.6.

X3J13 voted in June 1989 to clarify that `file-author` accepts only non-wild pathnames; an error is signaled if `wild-pathname-p` would be true of the *file* argument.

X3J13 voted in June 1989 to require `file-author` to accept logical pathnames (see section 23.1.5).

[Function] **file-position** *file-stream* **&optional** *position*

`file-position` returns or sets the current position within a random-access file.

`(file-position` *`file-stream`*`)` returns a non-negative integer indicating the current position within the *file-stream*, or `nil` if this cannot be determined. The file position at the start of a file will be zero. The value returned by `file-position` increases monotonically as input or output operations are performed. For a character file, performing a single `read-char` or `write-char` operation may cause the file position to be increased by more than 1 because of character-set translations (such as translating between the Common Lisp `#\Newline` character and an external ASCII carriage-return/line-feed sequence) and other aspects of the implementation. For a binary file, every `read-byte` or `write-byte` operation increases the file position by 1.

`(file-position` *`file-stream position`*`)` sets the position within *file-stream* to be *position*. The *position* may be an integer, or `:start` for the beginning of the stream, or `:end` for the end of the stream. If the integer is too large or otherwise inappropriate, an error is signaled (the `file-length` function returns the length beyond which `file-position` may not access). An integer returned by `file-position` of one argument should, in general, be acceptable as a second argument for use with the same file. With two arguments, `file-position` returns `t` if the repositioning was performed successfully, or `nil` if it was not (for example, because the file was not random-access).

Заметка для реализации: Implementations that have character files represented as a sequence of records of bounded size might choose to encode the file position as, for example, *record-number**256+*character-within-record*. This is a valid encoding because it increases monotonically as each character is read or written, though not necessarily by 1 at each step. An integer might then be considered "inappropriate" as a second argument to `file-position` if, when decoded into record number and character number, it turned out that the specified record was too short for the specified character number.

[Function] **file-length** *file-stream*

file-stream must be a stream that is open to a file. The length of the file is returned as a non-negative integer, or `nil` if the length cannot be determined. For a binary file, the length is specifically measured in units of the `:element-type` specified when the file was opened (see `open`).

[Function] **file-string-length** *file-stream object*

X3J13 voted in June 1989 to add the function `file-string-length`. The *object* must be a string or a character. The function `file-string-length` returns a non-negative integer that is the difference between what the `file-position` of the *file-stream* would be after and before writing the *object* to the *file-stream*, or `nil` if this difference cannot be determined. The value returned may depend on the current state of the *file-stream*; that is, calling `file-string-length` on the same arguments twice may in certain circumstances produce two different integers.

23.4 Loading Files

To *load* a file is to read through the file, evaluating each form in it. Programs are typically stored in files containing calls to constructs such as `defun`, `defmacro`, and `defvar`, which define the functions and variables of the program.

Loading a compiled ("fasload") file is similar, except that the file does not contain text but rather pre-digested expressions created by the compiler that can be loaded more quickly.

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

This function loads the file named by *filename* into the Lisp environment. It is assumed that a text (character file) can be automatically distinguished from an object (binary) file by some appropriate implementation-dependent means, possibly by the file type. The defaults for *filename* are taken from the variable `*default-pathname-defaults*`. If the *filename* (after the merging in of the defaults) does not explicitly specify a type, and both text and object types of the file are available in the file system, `load` should try to select the more appropriate file by some implementation-dependent means.

If the first argument is a stream rather than a pathname, then `load` determines what kind of stream it is and loads directly from the stream.

The `:verbose` argument (which defaults to the value of `*load-verbose*`), if true, permits `load` to print a message in the form of a comment (that is, with a leading semicolon) to `*standard-output*` indicating what file is being loaded and other useful information.

The `:print` argument (default `nil`), if true, causes the value of each expression loaded to be printed to `*standard-output*`. If a binary file is being loaded, then what is printed may not reflect precisely the contents of the source file, but nevertheless some information will be printed. X3J13 voted in March 1989 to add the variable `*load-print*`; its value is used as the default for the `:print` argument to `load`.

The function `load` rebinds `*package*` to its current value. If some form in the file changes the value of `*package*` during loading, the old value will be restored when the loading is completed. (This was specified in the first edition under the description of `*package*`; for convenience I now mention it here as well.)

X3J13 voted in March 1988 to specify exactly which streams may be used as pathnames. See section 23.1.6.

X3J13 voted in June 1989 to clarify that supplying a wild pathname as the *filename* argument to `load` has implementation-dependent consequences; `load` might signal an error, for example, or might load all files that match the pathname.

X3J13 voted in June 1989 to require `load` to accept logical pathnames (see section 23.1.5).

If a file is successfully loaded, `load` always returns a non-`nil` value. If `:if-does-not-exist` is specified and is `nil`, `load` just returns `nil` rather than signaling an error if the file does not exist.

X3J13 voted in March 1989 to require that `load` bind `*readtable*` to its current value at the time `load` is called; the dynamic extent of the binding should encompass all of the file-loading activity. This allows a portable program to include forms such as

```
(in-package "FOO")

(eval-when (:execute :load-toplevel :compile-toplevel)
  (setq *readtable* foo:my-readtable))
```

without performing a net global side effect on the loading environment. Such statements allow the remainder of such a file to be read either as interpreted code or by `compile-file` in a syntax determined by an alternative readtable.

X3J13 voted in June 1989 to require that `load` bind two new variables `*load-pathname*` and `*load-truename*`; the dynamic extent of the bindings should encompass all of the file-loading activity.

[Variable] ***load-verbose***

This variable provides the default for the `:verbose` argument to `load`. Its initial value is implementation-dependent.

[Variable] ***load-print***

X3J13 voted in March 1989 to add `*load-print*`. This variable provides the default for the `:print` argument to `load`. Its initial value is `nil`.

[Variable] ***load-pathname***

X3J13 voted in June 1989 to introduce `*load-pathname*`; it is initially `nil` but `load` binds it to a pathname that represents the file name given as the first argument to `load` merged with the defaults (see `merge-pathname`).

[Variable] ***load-truename***

X3J13 voted in June 1989 to introduce `*load-truename*`; it is initially `nil` but `load` binds it to the "true name" of the file being loaded. See `truename`.

X3J13 voted in March 1989 to introduce a facility based on the Object System whereby a user can specify how `compile-file` and `load` must cooperate to reconstruct compile-time constant objects at load time. The protocol is simply this: `compile-file` calls the generic function `make-load-form` on any object that is referenced as a constant or as a self-evaluating form, if the object's metaclass is `standard-class`, `structure-class`, any user-defined metaclass (not a subclass of `built-in-class`), or any of a possibly empty implementation-defined list of other metaclasses; `compile-file` will call `make-load-form` only once for any given object (as determined by `eq`) within a single file. The user-programmability stems from the possibility of user-defined methods for `make-load-form`. The helper function `make-load-form-saving-slots` makes it easy to write commonly used versions of such methods.

[Generic function] **make-load-form** *object*

The argument is an object that is referenced as a constant or as a self-evaluating form in a file being compiled by `compile-file`. The objective is to enable `load` to construct an equivalent object.

The first value, called the *creation form*, is a form that, when evaluated at load time, should return an object that is equivalent to the argument. The exact meaning of "equivalent" depends on the type of object and is up to the programmer who defines a method for `make-load-form`. This allows the user to program the notion of "similar as a constant" (see section 24.1).

The second value, called the *initialization form*, is a form that, when evaluated at load time, should perform further initialization of the object. The value returned by the initialization form is ignored. If the `make-load-form` method returns only one value, the initialization form is `nil`, which has no effect. If the object used as the argument to `make-load-form` appears as a constant in the initialization form, at load time it will be replaced by the equivalent object constructed by the creation form; this is how the further initialization gains access to the object.

Two values are returned so that circular structures may be handled. The order of evaluation rules discussed below for creation and initialization forms eliminates the possibility of partially initialized objects in the absence of circular structures and reduces the possibility to a minimum in the presence of circular structures. This allows nodes in non-circular structures to be built out of fully initialized subparts.

Both the creation form and the initialization form can contain references to objects of user-defined types (defined precisely below). However, there must not be any circular dependencies in creation forms. An example of a circular dependency: the creation form for the object X contains a reference to the object Y, and the creation form for the object Y contains a reference to the object X. A simpler example: the creation form for the object X contains a reference to X itself. Initialization forms are not subject to any restriction against circular dependencies, which is the entire reason for having initialization forms. See the example of circular data structures below.

The creation form for an object is always evaluated before the initialization form for that object. When either the creation form or the initialization form refers to other objects of user-defined types that have not been referenced earlier in the `compile-file`, the compiler collects all of the creation and initialization forms. Each initialization form is evaluated as soon as pos-

sible after its creation form, as determined by data flow. If the initialization form for an object does not refer to any other objects of user-defined types that have not been referenced earlier in the `compile-file`, the initialization form is evaluated immediately after the creation form. If a creation or initialization form F references other objects of user-defined types that have not been referenced earlier in the `compile-file`, the creation forms for those other objects are evaluated before F and the initialization forms for those other objects are also evaluated before F whenever they do not depend on the object created or initialized by F. Where the above rules do not uniquely determine an order of evaluation, it is unspecified which of the possible orders of evaluation is chosen.

While these creation and initialization forms are being evaluated, the objects are possibly in an uninitialized state, analogous to the state of an object between the time it has been created by `allocate-instance` and it has been processed fully by `initialize-instance`. Programmers writing methods for `make-load-form` must take care in manipulating objects not to depend on slots that have not yet been initialized.

It is unspecified whether `load` calls `eval` on the forms or does some other operation that has an equivalent effect. For example, the forms might be translated into different but equivalent forms and then evaluated; they might be compiled and the resulting functions called by `load` (after they themselves have been loaded); or they might be interpreted by a special-purpose interpreter different from `eval`. All that is required is that the effect be equivalent to evaluating the forms.

It is valid for user programs to call `make-load-form` in circumstances other than compilation, providing the argument's metaclass is not `built-in-class` or a subclass of `built-in-class`.

Applying `make-load-form` to an object whose metaclass is `standard-class` or `structure-class` for which no user-defined method is applicable signals an error. It is valid to implement this either by defining default methods for the classes `standard-object` and `structure-object` that signal an error or by having no applicable method for those classes.

See `load-time-eval`.

In the following example, an equivalent instance of `my-class` is reconstructed by using the values of two of its slots. The value of the third slot is derived from those two values.

```
(defclass my-class ()   ((a :initarg :a :reader my-a)
  (b :initarg :b :reader my-b)
  (c :accessor my-c)))

(defmethod shared-initialize ((self my-class) slots &rest inits)
 (declare (ignore slots inits))
 (unless (slot-boundp self 'c)
   (setf (my-c self)
         (some-computation (my-a self) (my-b self)))))

(defmethod make-load-form ((self my-class))
 `(make-instance ',(class-name (class-of self))
                 :a ',(my-a self) :b ',(my-b self)))
```

This code will fail if either of the first two slots of some instance of `my-class` contains the instance itself. Another way to write the last form in the preceding example is

```
(defmethod make-load-form ((self my-class))
 (make-load-form-saving-slots self '(a b)))
```

This has the advantages of conciseness and handling circularities correctly.

In the next example, instances of class `my-frob` are "interned" in some way. An equivalent instance is reconstructed by using the value of the `name` slot as a key for searching for existing objects. In this case the programmer has chosen to create a new object if no existing object is found; an alternative possibility would be to signal an error in that case.

```
(defclass my-frob ()
 ((name :initarg :name :reader my-name)))

(defmethod make-load-form ((self my-frob))
 `(find-my-frob ',(my-name self) :if-does-not-exist :create))
```

In the following example, the data structure to be dumped is circular, because each node of a tree has a list of its children and each child has a reference back to its parent.

```
(defclass tree-with-parent () ((parent :accessor tree-parent)
                               (children :initarg :children)))
```

```
(defmethod make-load-form ((x tree-with-parent))
  (values
    `(make-instance ',(class-of x)
                    :children ',(slot-value x 'children))
    `(setf (tree-parent ',x) ',(slot-value x 'parent))))
```

Suppose `make-load-form` is called on one object in such a structure. The creation form creates an equivalent object and fills in the `children` slot, which forces creation of equivalent objects for all of its children, grandchildren, etc. At this point none of the parent slots have been filled in. The initialization form fills in the `parent` slot, which forces creation of an equivalent object for the parent if it was not already created. Thus the entire tree is recreated at load time. At compile time, `make-load-form` is called once for each object in the tree. All the creation forms are evaluated, in unspecified order, and then all the initialization forms are evaluated, also in unspecified order.

In this final example, the data structure to be dumped has no special properties and an equivalent structure can be reconstructed simply by reconstructing the slots' contents.

```
(defstruct my-struct a b c)
(defmethod make-load-form ((s my-struct))
  (make-load-form-saving-slots s))
```

This is easy to code using `make-load-form-saving-slots`.

[Function] **make-load-form-saving-slots** *object* **&optional** *slots*

This returns two values suitable for return from a `make-load-form` method. The first argument is the object. The optional second argument is a list of the names of slots to preserve; it defaults to all of the local slots.

`make-load-form-saving-slots` returns forms that construct an equivalent object using `make-instance` and `setf` of `slot-value` for slots with values, or `slot-makunbound` for slots without values, or other functions of equivalent effect.

Because `make-load-form-saving-slots` returns two values, it can deal with circular structures; it works for any object of metaclass `standard-class` or `structure-class`. Whether the result is useful depends on whether the object's type and slot contents fully capture an application's idea of the object's state.

23.5 Accessing Directories

The following function is a very simple portable primitive for examining a directory. Most file systems can support much more powerful directory-searching primitives, but no two are alike. It is expected that most implementations of Common Lisp will extend the `directory` function or provide more powerful primitives.

[Function] **directory** *pathname* **&key**

A list of pathnames is returned, one for each file in the file system that matches the given *pathname*. (The *pathname* argument may be a pathname, a string, or a stream associated with a file.) For a file that matches, the `truename` appears in the result list. If no file matches the *pathname*, it is not an error; `directory` simply returns `nil`, the list of no results. Keywords such as `:wild` and `:newest` may be used in *pathname* to indicate the search space.

X3J13 voted in March 1988 to specify exactly which streams may be used as pathnames. See section 23.1.6.

X3J13 voted in January 1989 to specify that `directory` is unaffected by whether the first argument, if a stream, is open or closed. If the first argument is a stream, `directory` behaves as if the function `pathname` were applied to the stream and the resulting pathname used instead. However, X3J13 commented that the treatment of open streams may differ considerably from one implementation to another; for example, in some operating systems open files are written under a temporary or invisible name and later renamed when closed. In general, programmers writing code intended to be portable should be careful when using `directory`.

X3J13 voted in June 1989 to require `directory` to accept logical pathnames (see section 23.1.5). However, the result returned by `directory` never contains a logical pathname.

Заметка для реализации: It is anticipated that an implementation may need to provide additional parameters to control the directory search. Therefore `directory` is specified to take additional keyword arguments so that implementations may experiment with extensions, even though no particular keywords are specified here.

As a simple example of such an extension, for a file system that supports the notion of cross-directory file links, a keyword argument `:links` might, if non-`nil`, specify that such links be included in the result list.

[Function] **ensure-directories-exist** *file* **&key** *:verbose*

If the directories of file do not exist then this function creates them returning two values, file and a second value true if the directories were created or nil if not.

Глава 24

Miscellaneous Features Разнообразные дополнительные возможности

In this chapter are described various things that don't seem to fit neatly anywhere else in this book: the compiler, the `documentation` function, debugging aids, environment inquiries (including facilities for calculating and measuring time), and the `identity` function.

В этой главе описываются различные вещи, которые не могли быть описаны в каком-либо другом месте книги: компилятор, функция поиска документации `documentation`, отладочный функционал, информация о среде выполнения (включая подсчет и измерение времени) и функция идентичности `identity`.

24.1 The Compiler Компилятор

The compiler is a program that may make code run faster by translating programs into an implementation-dependent form that can be executed more efficiently by the computer. Most of the time you can write programs without worrying about the compiler; compiling a file of code should produce an equivalent but more efficient program. When doing more esoteric things, you may need to think carefully about what happens at "compile time" and what happens at "load time." Then the `eval-when` construct becomes particularly useful.

Компилятор — это программа, которая выполняет преобразование кода для того, чтобы его выполнение было более быстрым. Способ преобразования зависит от реализации. Обычно вы можете писать программы, не беспокоясь о компиляторе. Компиляция файла с кодом должна создавать эквивалентную, но более быструю программу. При выполнении эзотерических вещей, вам необходимо подумать о том, что случается во «время компиляции» и, что случает во «время загрузки». В таких случаях бывает полезна конструкция `eval-when`.

Most declarations are not used by the Common Lisp interpreter; they may be used to give advice to the compiler. The compiler may attempt to check your advice and warn you if it is inconsistent.

Большинство деклараций не используются интерпретатором Common Lisp'a, они могут использования для советов компилятору. Компилятор может попытаться проверить ваш совет и уведомить вас, если совет противоречивый.

Unlike most other Lisp dialects, Common Lisp recognizes `special` declarations in interpreted code as well as compiled code.

В отличие от других диалектов Lisp'a, Common Lisp распознает декларацию `special` как в интерпретируемом, так и в скомпилированном коде.

The internal workings of a compiler will of course be highly implementation-dependent. The following functions provide a standard interface to the compiler, however.

Внутренняя работа компилятора конечно же будет будет во многом зависеть от конкретной реализации. Однако, следующие функции предоставляют стандартный интерфейс доступа к компилятору.

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

If *definition* is supplied, it should be a lambda-expression, the interpreted function to be compiled. If it is not supplied, then *name* should be a symbol with a definition that is a lambda-expression; that definition is compiled and the resulting compiled code is put back into the symbol as its function definition.

name may be any function-name (a symbol or a list whose car is `setf`—see section 7.1). One may write `(compile '(setf cadr))` to compile the `setf` expansion function for `cadr`.

If the optional *definition* argument is supplied, it may be either a lambda-expression (which is coerced to a function) or a function to be compiled; if no *definition* is supplied, the `symbol-function` of the symbol is extracted and compiled. It is permissible for the symbol to have a macro definition rather than a function definition; both macros and functions may be compiled.

It is an error if the function to be compiled was defined interpretively in a non-null lexical environment. (An implementation is free to extend the behavior of `compile` to compile such functions properly, but portable programs may not depend on this capability.) The consequences of calling `compile` on a function that is already compiled are unspecified.

The definition is compiled and a compiled-function object produced. If *name* is a non-`nil` symbol, then the compiled-function object is installed as the global function definition of the symbol and the symbol is returned. If *name* is `nil`, then the compiled-function object itself is returned. For example:

```
(defun foo ...) ⇒ foo       ;A function definition
(compile 'foo) ⇒ foo        ;Compile it
                            ;Now foo runs faster (maybe)

(compile nil
      '(lambda (a b c) (- (* b b) (* 4 a c))))
  ⇒ a compiled function of three arguments that computes b^2 − 4ac
```

X3J13 voted in June 1989 to specify that `compile` returns two additional values indicating whether the compiler issued any diagnostics (see section 24.1.1).

X3J13 voted in March 1989 to add two new keyword arguments `:verbose` and `:print` to `compile-file` by analogy with `load`. The new function definition is as follows.

[Function] **compile-file** *input-pathname* **&key** *:output-file :verbose :print*

The *input-pathname* must be a valid file specifier, such as a pathname. The defaults for *input-filename* are taken from the variable `*default-pathname-defaults*`. The file should be a Lisp source file; its contents are compiled and written as a binary object file.

The `:verbose` argument (which defaults to the value of `*compile-verbose*`), if true, permits `compile-file` to print a message in the form of a comment to `*standard-output*` indicating what file is being compiled and other useful information.

The `:print` argument (which defaults to the value of `*compile-print*`), if true, causes information about top-level forms in the file being compiled to be printed to `*standard-output*`. Exactly what is printed is implementation-dependent; nevertheless something will be printed.

X3J13 voted in March 1988 to specify exactly which streams may be used as pathnames (see section 23.1.6). X3J13 voted in June 1989 to clarify that supplying a wild pathname as the *input-pathname* argument to `compile-file` has implementation-dependent consequences; `compile-file` might signal an error, for example, or might compile all files that match the wild pathname.

X3J13 voted in June 1989 to require `compile-file` to accept logical pathnames (see section 23.1.5).

The `:output-file` argument may be used to specify an output pathname; it defaults in a manner appropriate to the implementation's file system conventions.

X3J13 voted in June 1989 to specify that `compile-file` returns three values: the `truename` of the output file (or `nil` if the file could not be created) and two values indicating whether the compiler issued any diagnostics (see section 24.1.1).

X3J13 voted in October 1988 to specify that `compile-file`, like `load`, rebinds `*package*` to its current value. If some form in the file changes the value of `*package*`, the old value will be restored when compilation is completed.

X3J13 voted in June 1989 to specify restrictions on conforming programs to ensure consistent handling of symbols and packages.

In order to guarantee that compiled files can be loaded correctly, the user must ensure that the packages referenced in the file are defined consistently at compile and load time. Conforming Common Lisp programs must satisfy the following requirements.

- The value of `*package*` when a top-level form in the file is processed by `compile-file` must be the same as the value of `*package*` when the code corresponding to that top-level form in the compiled file is executed by the loader. In particular, any top-level form in a file that

alters the value of `*package*` must change it to a package of the same name at both compile and load time; moreover, if the first non-atomic top-level form in the file is not a call to `in-package`, then the value of `*package*` at the time `load` is called must be a package with the same name as the package that was the value of `*package*` at the time `compile-file` was called.

- For every symbol appearing lexically within a top-level form that was accessible in the package that was the value of `*package*` during processing of that top-level form at compile time, but whose home package was another package, at load time there must be a symbol with the same name that is accessible in both the load-time `*package*` and in the package with the same name as the compile-time home package.
- For every symbol in the compiled file that was an external symbol in its home package at compile time, there must be a symbol with the same name that is an external symbol in the package with the same name at load time.

If any of these conditions do not hold, the package in which `load` looks for the affected symbols is unspecified. Implementations are permitted to signal an error or otherwise define this behavior.

These requirements are merely an explicit statement of the status quo, namely that users cannot depend on any particular behavior if the package environment at load time is inconsistent with what existed at compile time.

X3J13 voted in March 1989 to specify that `compile-file` must bind `*readtable*` to its current value at the time `compile-file` is called; the dynamic extent of the binding should encompass all of the file-loading activity. This allows a portable program to include forms such as

```
(in-package "FOO")

(eval-when (:execute :load-toplevel :compile-toplevel)
  (setq *readtable* foo:my-readtable))
```

without performing a net global side effect on the loading environment. Such statements allow the remainder of such a file to be read either as interpreted code or by `compile-file` in a syntax determined by an alternative readtable.

X3J13 voted in June 1989 to require that `compile-file` bind two new variables `*compile-file-pathname*` and `*compile-file-truename*`; the dynamic extent of the bindings should encompass all of the file-compiling activity.

[Variable] ***compile-verbose***

This variable provides the default for the `:verbose` argument to `compile-file`. Its initial value is implementation-dependent.

[Variable] ***compile-print***

This variable provides the default for the `:print` argument to `compile-file`. Its initial value is implementation-dependent.

[Variable] ***compile-file-pathname***

X3J13 voted in June 1989 to introduce `*compile-file-pathname*`; it is initially `nil` but `compile-file` binds it to a pathname that represents the file name given as the first argument to `compile-file` merged with the defaults (see `merge-pathname`).

[Variable] ***compile-file-truename***

Variable is initially `nil` but `compile-file` binds it to the "true name" of the pathname of the file being compiled. See `truename`.

[Специальный оператор] **load-time-value** form [read-only-p]

This is a mechanism for delaying evaluation of a *form* until it can be done in the run-time environment.

If a `load-time-value` expression is seen by `compile-file`, the compiler performs its normal semantic processing (such as macro expansion and translation into machine code) on the form, but arranges for the execution of the *form* to occur at load time in a null lexical environment, with the result of this evaluation then being treated as an immediate quantity (that is, as if originally quoted) at run time. It is guaranteed that the evaluation of the *form* will take place only once when the file is loaded, but the order of evaluation with respect to the execution of top-level forms in the file is unspecified.

If a `load-time-value` expression appears within a function compiled with `compile`, the *form* is evaluated at compile time in a null lexical environment. The result of this compile-time evaluation is treated as an immediate quantity in the compiled code.

In interpreted code, *form* is evaluated (by `eval`) in a null lexical environment and one value is returned. Implementations that implicitly compile (or partially compile) expressions passed to `eval` may evaluate the *form* only once, at the time this compilation is performed. This is intentionally similar to the freedom that implementations are given for the time of expanding macros in interpreted code.

If the same (as determined by `eq`) list `(load-time-value` *`form`*`)` is evaluated or compiled more than once, it is unspecified whether the *form* is evaluated only once or is evaluated more than once. This can happen both when an expression being evaluated or compiled shares substructure and when the same expression is passed to `eval` or to `compile` multiple times. Since a `load-time-value` expression may be referenced in more than one place and may be evaluated multiple times by the interpreter, it is unspecified whether each execution returns a "fresh" object or returns the same object as some other execution. Users must use caution when destructively modifying the resulting object.

If two lists `(load-time-value` *`form`*`)` are `equal` but not `eq`, their values always come from distinct evaluations of *form*. Coalescing of these forms is not permitted.

The optional *read-only-p* argument designates whether the result may be considered a read-only constant. If `nil` (the default), the result must be considered ordinary, modifiable data. If `t`, the result is a read-only quantity that may, as appropriate, be copied into read-only space and may, as appropriate, be shared with other programs. The *read-only-p* argument is not evaluated and only the literal symbols `t` and `nil` are permitted.

This new feature addresses the same set of needs as the now-defunct `#,` reader syntax but in a cleaner and more general manner. Note that `#,` syntax was reliably useful only inside quoted structure (though this was not explicitly mentioned in the first edition), whereas a `load-time-value` form must appear outside quoted structure in a for-evaluation position.

See `make-load-form`.

[Function] **disassemble** *name-or-compiled-function*

The argument should be a function object, a lambda-expression, or a symbol with a function definition. If the relevant function is not a compiled function, it is first compiled. In any case, the compiled code is then "reverse-assembled" and printed out in a symbolic format. This is primarily useful for debugging the compiler, but also often of use to the novice who wishes to understand the workings of compiled code.

Заметка для реализации: Implementors are encouraged to make the output readable, preferably with helpful comments.

When `disassemble` compiles a function, it never installs the resulting compiled-function object in the `symbol-function` of a symbol.

name may be any function-name (a symbol or a list whose car is `setf`—see section 7.1). Thus one may write `(disassemble '(setf cadr))` to disassemble the `setf` expansion function for `cadr`.

[Function] **function-lambda-expression** *fn*

This function allows the source code for a defined function to be recovered. (The committee noted that the first edition provided no portable way to recover a lambda-expression once it had been compiled or evaluated to produce a function.)

This function takes one argument, which must be a function, and returns three values.

The first value is the defining lambda-expression for the function, or `nil` if that information is not available. The lambda-expression may have been preprocessed in some ways but should nevertheless be of a form suitable as an argument to the function `compile` or for use in the `function` special operator.

The second value is `nil` if the function was definitely produced by closing a lambda-expression in the null lexical environment; it is some non-`nil` value if the function might have been closed in some non-null lexical environment.

The third value is the "name" of the function; this is `nil` if the name is not available or if the function had no name. The name is intended for debugging purposes only and may be any Lisp object (not necessarily one that would be valid for use as a name in a `defun` or `function` special operator, for example).

Заметка для реализации: An implementation is always free to return the values `nil`, `t`, `nil` from this function but is encouraged to make more useful in-

formation available as appropriate. For example, it may not be desirable for files of compiled code to retain the source lambda-expressions for use after the file is loaded, but it is probably desirable for functions produced by "in-core" calls to `eval`, `compile`, or `defun` to retain the defining lambda-expression for debugging purposes. The function `function-lambda-expression` makes this information, if retained, accessible in a standard and portable manner.

[Макрос] **with-compilation-unit** ({option-name option-value}*) {form}*

`with-compilation-unit` executes the body forms as an implicit `progn`. Within the dynamic context of this form, warnings deferred by the compiler until "the end of compilation" will be deferred until the end of the outermost call to `with-compilation-unit`. The results are the same as those of the last of the forms (or `nil` if there is no *form*).

Each *option-name* is an unevaluated keyword; each *option-value* is evaluated. The set of keywords permitted may be extended by the implementation, but the only standard option keyword is `:override`; the default value for this option is `nil`. If `with-compilation-unit` forms are nested dynamically, only the outermost such call has any effect unless the `:override` value of an inner call is true.

The function `compile-file` should provide the effect of

(with-compilation-unit (:override nil) ...)

around its code.

Any implementation-dependent extensions to this behavior may be provided only as the result of an explicit programmer request by use of an implementation-dependent keyword. It is forbidden for an implementation to attach additional meaning to a conforming use of this macro.

Note that not all compiler warnings are deferred. In some implementations, it may be that none are deferred. This macro only creates an interface to the capability where it exists, it does not require the creation of the capability. An implementation that does not defer any compiler warnings may correctly implement this macro as an expansion into a simple `progn`.

24.1.1 Compiler Diagnostics

`compile` and `compile-file` may output warning messages; any such messages should go to the stream that is the value of `*error-output*`.

First, note that `error` and `warning` conditions may be signaled either by the compiler itself or by code being processed by the compiler (for example, arbitrary errors may occur during compile-time macro expansion or processing of `eval-when` forms). Considering only those conditions signaled *by the compiler* (as opposed to *during compilation*):

- Conditions of type `error` may be signaled by the compiler in situations where the compilation cannot proceed without intervention. Examples of such situations may include errors when opening a file or syntax errors.

- Conditions of type `warning` may be signaled by the compiler in situations where the standard explicitly states that a warning must, should, or may be signaled. They may also be signaled when the compiler can determine that a situation would result at runtime that would have undefined consequences or would cause an error to be signaled. Examples of such situations may include violations of type declarations, altering or rebinding a constant defined with `defconstant`, calls to built-in Lisp functions with too few or too many arguments or with malformed keyword argument lists, referring to a variable declared `ignore`, or unrecognized declaration specifiers.

- The compiler is permitted to signal diagnostics about matters of programming style as conditions of type `style-warning`, a subtype of `warning`. Although a `style-warning` condition *may* be signaled in these situations, no implementation is *required* to do so. However, if an implementation does choose to signal a condition, that condition will be of type `style-warning` and will be signaled by a call to the function `warn`. Examples of such situations may include redefinition of a function with an incompatible argument list, calls to functions (other than built-in functions) with too few or too many arguments or with malformed keyword argument lists, unreferenced local variables not declared `ignore`, or standard declaration specifiers that are ignored by the particular compiler in question.

Both `compile` and `compile-file` are permitted (but not required) to establish a handler for conditions of type `error`. Such a handler might, for example, issue a warning and restart compilation from some implementation-dependent point in order to let the compilation proceed without manual intervention.

The functions `compile` and `compile-file` each return three values. See the definitions of these functions for descriptions of the first value. The second value is `nil` if no compiler diagnostics were issued, and true otherwise. The third value is `nil` if no compiler diagnostics other than style warnings were issued; a non-`nil` value indicates that there were "serious" compiler diagnostics issued or that other conditions of type `error` or `warning` (but not `style-warning`) were signaled during compilation.

24.1.2 Compiled Functions

Certain requirements are imposed on the functions produced by the compilation process.

If a function is of type `compiled-function`, then all macro calls appearing lexically within the function have already been expanded and will not be expanded again when the function is called. The process of compilation effectively turns every `macrolet` or `symbol-macrolet` construct into a `progn` (or a `locally`) with all instances of the local macros in the body fully expanded.

If a function is of type `compiled-function`, then all `load-time-value` forms appearing lexically within the function have already been pre-evaluated and will not be evaluated again when the function is called.

Implementations are free to classify every function as a `compiled-function` provided that all functions satisfy the preceding requirements. Conversely, it is permissible for a function that is not a `compiled-function` to satisfy the preceding requirements.

If one or more functions are defined in a file that is compiled with `compile-file` and the compiled file is subsequently loaded by the function `load`, the resulting loaded function definitions must be of type `compiled-function`.

The function `compile` must produce an object of type `compiled-function` as the value that is either returned or stored into the `symbol-function` of a symbol argument.

Note that none of these restrictions addresses questions of the compilation technology or target instruction set. For example, a compiled function does not necessarily consist of native machine instructions. These requirements merely specify the behavior of the type system with respect to certain actions taken by `compile`, `compile-file`, and `load`.

24.1.3 Compilation Environment

Following information must be available at compile time for correct compilation and what need not be available until run time.

The following information must be present in the compile-time environment for a program to be compiled correctly. This information need not also be present in the run-time environment.

- In conforming code, macros referenced in the code being compiled must have been previously defined in the compile-time environment. The compiler must treat as a function call any form that is a list whose *car* is a symbol that does not name a macro or special operator. (This implies that `setf` methods must also be available at compile time.)

- In conforming code, proclamations for `special` variables must be made in the compile-time environment before any bindings of those variables are processed by the compiler. The compiler must treat any binding of an undeclared variable as a lexical binding.

The compiler may incorporate the following kinds of information into the code it produces, if the information is present in the compile-time environment and is referenced within the code being compiled; however, the compiler is not required to do so. When compile-time and run-time definitions differ, it is unspecified which will prevail within the compiled code (unless some other behavior is explicitly specified below). It is also permissible for an implementation to signal an error at run time on detecting such a discrepancy. In all cases, the absence of the information at compile time is not an error, but its presence may enable the compiler to generate more efficient code.

- The compiler may assume that functions that are defined and declared `inline` in the compile-time environment will retain the same definitions at run time.

- The compiler may assume that, within a named function, a recursive call to a function of the same name refers to the same function, unless that function has been declared `notinline`. (This permits tail-recursive calls of a function to itself to be compiled as jumps, for example, thereby turning certain recursive schemas into efficient loops.)

- In the absence of `notinline` declarations to the contrary, `compile-file` may assume that a call within the file being compiled to a named function that is defined in that file refers to that function. (This rule permits *block compilation* of files.) The behavior of the program is unspecified if functions are redefined individually at run time.

- The compiler may assume that the signature (or "interface contract") of all built-in Common Lisp functions will not change. In addition, the compiler may treat all built-in Common Lisp functions as if they had been proclaimed `inline`.

- The compiler may assume that the signature (or "interface contract") of functions with `ftype` information available will not change.

- The compiler may "wire in" (that is, open-code or inline) the values of symbolic constants that have been defined with `defconstant` in the compile-time environment.

- The compiler may assume that any type definition made with `defstruct` or `deftype` in the compile-time environment will retain the same definition in the run-time environment. It may also assume that a class defined by `defclass` in the compile-time environment will be defined in the run-time environment in such a way as to have the same superclasses and metaclass. This implies that subtype/supertype relationships of type specifiers will not change between compile time and run time. (Note that it is not an error for an unknown type to appear in a declaration at compile time, although it is reasonable for the compiler to emit a warning in such a case.)

- The compiler may assume that if type declarations are present in the compile-time environment, the corresponding variables and functions present in the run-time environment will actually be of those types. If this assumption is violated, the run-time behavior of the program is undefined.

The compiler must not make any additional assumptions about consistency between the compile-time and run-time environments. In particular, the compiler may not assume that functions that are defined in the compile-time environment will retain either the same definition or the same signature at run time, except as described above. Similarly, the compiler may not signal an error if it sees a call to a function that is not defined at compile time, since that function may be provided at run time.

X3J13 voted in January 1989 to specify the compile-time side effects of processing various macro forms.

Calls to defining macros such as `defmacro` or `defvar` appearing within a file being processed by `compile-file` normally have compile-time side effects that affect how subsequent forms in the same file are compiled. A convenient model for explaining how these side effects happen is that each defining macro expands into one or more `eval-when` forms and that compile-time side effects are caused by calls occurring in the body of an `(eval-when (:compile-toplevel) ...)` form.

The affected defining macros and their specific side effects are as follows. In each case, it is identified what a user must do to ensure that a program is conforming, and what a compiler must do in order to correctly process a conforming program.

`deftype` The user must ensure that the body of a `deftype` form is evaluable at compile time if the type is referenced in subsequent type declarations. The compiler must ensure that a type specifier defined by `deftype` is recognized in subsequent type declarations. If the expansion of a type specifier is not defined fully at compile time (perhaps because it expands into an unknown type specifier or a `satisfies` of a named function that isn't defined in the compile-time environment), an implementation may ignore any references to this type in declarations and may signal a warning.

`defmacro` **and** `define-modify-macro` The compiler must store macro definitions at compile time, so that occurrences of the macro later on in the file can be expanded correctly. The user must ensure that the body of the macro is evaluable at compile time if it is referenced within the file being compiled.

`defun` No required compile-time side effects are associated with `defun` forms.

In particular, `defun` does not make the function definition available at compile time. An implementation may choose to store information about the function for the purposes of compile-time error checking (such as checking the number of arguments on calls) or to permit later `inline` expansion of the function.

`defvar` **and** `defparameter` The compiler must recognize that the variables named by these forms have been proclaimed `special`. However, it must not evaluate the *initial-value* form or `set` the variable at compile time.

`defconstant` The compiler must recognize that the symbol names a constant. An implementation may choose to evaluate the *value-form* at compile time, load time, or both. Therefore the user must ensure that the *value-form* is evaluable at compile time (regardless of whether or not references to the constant appear in the file) and that it always evaluates to the same value. (There has been considerable variance among implementations on this point. The effect of this specification is to legitimize all of the implementation variants by requiring care of the user.)

`defsetf` **and** `define-setf-method` The compiler must make `setf` methods available so that they may be used to expand calls to `setf` later on in the file. Users must ensure that the body of a call to `define-setf-method` or the complex form of `defsetf` is evaluable at compile time if the corresponding place is referred to in a subsequent `setf` in the same file. The compiler must make these `setf` methods available to compile-time calls to `get-setf-method` when its environment argument is a value received as the `&environment` parameter of a macro.

`defstruct` The compiler must make the structure type name recognized as a valid type name in subsequent declarations (as described above for `deftype`) and make the structure slot accessors known to `setf`. In addition, the compiler must save enough information so that further `defstruct` definitions can include (with the `:include` option) a structure type defined earlier in the file being compiled. The functions that `defstruct` generates are not defined in the compile-time environment,

although the compiler may save enough information about the functions to allow `inline` expansion of subsequent calls to these functions. The `#S` reader syntax may or may not be available for that structure type at compile time.

`define-condition` The rules are essentially the same as those for `defstruct`. The compiler must make the condition type recognizable as a valid type name, and it must be possible to reference the condition type as the *parent-type* of another condition type in a subsequent `define-condition` form in the file being compiled.

`defpackage` All of the actions normally performed by the `defpackage` macro at load time must also be performed at compile time.

Compile-time side effects may cause information about a definition to be stored in a different manner from information about definitions processed either interpretively or by loading a compiled file. In particular, the information stored by a defining macro at compile time may or may not be available to the interpreter (either during or after compilation) or during subsequent calls to `compile` or `compile-file`. For example, the following code is not portable because it assumes that the compiler stores the macro definition of `foo` where it is available to the interpreter.

```
(defmacro foo (x) `(car ,x))

(eval-when (:execute :compile-toplevel :load-toplevel)
  (print (foo '(a b c))))      ;Wrong
```

The goal may be accomplished portably by including the macro definition within the `eval-when` form:

```
(eval-when (eval compile load)
  (defmacro foo (x) `(car ,x))
  (print (foo '(a b c))))      ;Right
```

`declaim` X3J13 voted in June 1989 to add a new macro `declaim` for making proclamations recognizable at compile time. The declaration specifiers in the `declaim` form are effectively proclaimed at compile time so as to affect compilation of subsequent forms. (Note that compiler processing of a call to `proclaim` does not have any compile-time side effects, for `proclaim` is a function.)

`in-package` X3J13 voted in March 1989 to specify that all of the actions normally performed by the `in-package` macro at load time must also be performed at compile time.

X3J13 voted in June 1989 to specify the compile-time side effects of processing various CLOS-related macro forms. Top-level calls to the CLOS defining macros have the following compile-time side effects; any other compile-time behavior is explicitly left unspecified.

`defclass` The class name may appear in subsequent type declarations and can be used as a specializer in subsequent `defmethod` forms. Thus the compile-time behavior of `defclass` is similar to that of `deftype` or `defstruct`.

`defgeneric` The generic function can be referenced in subsequent `defmethod` forms, but the compiler does not arrange for the generic function to be callable at compile time.

`defmethod` The compiler does not arrange for the method to be callable at compile time. If there is a generic function with the same name defined at compile time, compiling a `defmethod` form does not add the method to that generic function; the method is added to the generic function only when the `defmethod` form is actually executed.

The error-signaling behavior described in the specification of `defmethod` in chapter 27 (if the function isn't a generic function or if the lambda-list is not congruent) occurs only when the defining form is executed, not at compile time.

The forms in `eql` parameter specializers are evaluated when the `defmethod` form is executed. The compiler is permitted to build in knowledge about what the form in an `eql` specializer will evaluate to in cases where the ultimate result can be syntactically inferred without actually evaluating it.

`define-method-combination` The method combination can be used in subsequent `defgeneric` forms.

The body of a `define-method-combination` form is evaluated no earlier than when the defining macro is executed and possibly as late as generic function invocation time. The compiler may attempt to evaluate these forms at compile time but must not depend on being able to do so.

24.1.4 Similarity of Constants

Following paragraphs specifies what objects can be in compiled constants and what relationship there must be between a constant passed to the compiler and the one that is established by compiling it and then loading its file.

The key is a definition of an equivalence relationship called "similarity as constants" between Lisp objects. Code passed through the file compiler and then loaded must behave as though quoted constants in it are similar in this sense to quoted constants in the corresponding source code. An object may be used as a quoted constant processed by `compile-file` if and only if the compiler can guarantee that the resulting constant established by loading the compiled file is "similar as a constant" to the original. Specific requirements are spelled out below.

Some types of objects, such as streams, are not supported in constants processed by the file compiler. Such objects may not portably appear as constants in code processed with `compile-file`. Conforming implementations are required to handle such objects either by having the compiler or loader reconstruct an equivalent copy of the object in some implementation-specific manner or by having the compiler signal an error.

Of the types supported in constants, some are treated as aggregate objects. For these types, being similar as constants is defined recursively. We say that an object of such a type has certain "basic attributes"; to be similar

as a constant to another object, the values of the corresponding attributes of the two objects must also be similar as constants.

A definition of this recursive form has problems with any circular or infinitely recursive object such as a list that is an element of itself. We use the idea of depth-limited comparison and say that two objects are similar as constants if they are similar at all finite levels. This idea is implicit in the definitions below, and it applies in all the places where attributes of two objects are required to be similar as constants. The question of handling circular constants is the subject of a separate vote by X3J13 (see below).

The following terms are used throughout this section. The term *constant* refers to a quoted or self-evaluating constant, not a named constant defined by `defconstant`. The term *source code* is used to refer to the objects constructed when `compile-file` calls `read` (or the equivalent) and to additional objects constructed by macro expansion during file compilation. The term *compiled code* is used to refer to objects constructed by `load`.

Two objects are *similar as a constant* if and only if they are both of one of the types listed below and satisfy the additional requirements listed for that type.

`number` Two numbers are similar as constants if they are of the same type and represent the same mathematical value.

`character` Two characters are similar as constants if they both represent the same character. (The intent is that this be compatible with how `eql` is defined on characters.)

`symbol` X3J13 voted in June 1989 to define similarity as a constant for interned symbols. A symbol S appearing in the source code is similar as a constant to a symbol S' in the compiled code if their print names are similar as constants and either of the following conditions holds:

- S is accessible in `*package*` at compile time and S' is accessible in `*package*` at load time.
- S' is accessible in the package that is similar as a constant to the home package of symbol S.

The "similar as constants" relationship for interned symbols has nothing to do with `*readtable*` or how the function `read` would parse the characters in the print name of the symbol.

An uninterned symbol in the source code is similar as a constant to an uninterned symbol in the compiled code if their print names are similar as constants.

`package` A package in the source code is similar as a constant to a package in the compiled code if their names are similar as constants. Note that the loader finds the corresponding package object as if by calling `find-package` with the package name as an argument. An error is signaled if no package of that name exists at load time.

`random-state` We say that two `random-state` objects are *functionally equivalent* if applying `random` to them repeatedly always produces the same pseudo-random numbers in the same order.

Two random-states are similar as constants if and only if copies of them made via `make-random-state` are functionally equivalent. (Note that a constant `random-state` object cannot be used as the *state* argument to the function `random` because `random` performs a side effect on that argument.)

`cons` Two conses are similar as constants if the values of their respective *car* and *cdr* attributes are similar as constants.

`array` Two arrays are similar as constants if the corresponding values of each of the following attributes are similar as constants: for vectors (one-dimensional arrays), the `length` and `element-type` and the result of `elt` for all valid indices; for all other arrays, the `array-rank`, the result of `array-dimension` for all valid axis numbers, the `array-element-type`, and the result of `aref` for all valid indices. (The point of distinguishing vectors is to take any fill pointers into account.)

If the array in the source code is a `simple-array`, then the corresponding array in the compiled code must also be a `simple-array`, but if the array in the source code is displaced, has a fill pointer, or is adjustable, the corresponding array in the compiled code is permitted to lack any or all of these qualities.

`hash-table` Two hash tables are similar as constants if they meet three requirements. First, they must have the same test (for example, both are `eql` hash tables or both are `equal` hash tables). Second, there must

be a unique bijective correspondence between the keys of the two tables, such that the corresponding keys are similar as constants. Third, for all keys, the values associated with two corresponding keys must be similar as constants.

If there is more than one possible one-to-one correspondence between the keys of the two tables, it is unspecified whether the two tables are similar as constants. A conforming program cannot use such a table as a constant.

`pathname` Two pathnames are similar as constants if all corresponding pathname components are similar as constants.

`stream`, `readtable`, **and** `method` Objects of these types are not supported in compiled constants.

`function` X3J13 voted in June 1989 to specify that objects of type `function` are not supported in compiled constants.

`structure` **and** `standard-object` X3J13 voted in March 1989 to introduce a facility based on the Common Lisp Object System whereby a user can specify how `compile-file` and `load` must cooperate to reconstruct compile-time constant objects at load time (see `make-load-form`).

X3J13 voted in March 1989 to specify the circumstances under which constants may be coalesced in compiled code.

Suppose A and B are two objects used as quoted constants in the source code, and that A' and B' are the corresponding objects in the compiled code. If A' and B' are `eql` but A and B were not `eql`, then we say that A and B have been *coalesced* by the compiler.

An implementation is permitted to coalesce constants appearing in code to be compiled if and only if they are similar as constants, except that objects of type `symbol`, `package`, `structure`, or `standard-object` obey their own rules and may not be coalesced by a separate mechanism.

Обоснование: Objects of type `symbol` and `package` cannot be coalesced because the fact that they are named, interned objects means they are already as coalesced as it is useful for them to be. Uninterned symbols could perhaps be coalesced, but that was thought to be more dangerous than useful. Structures and objects could be coalesced if a "similar as a constant" predicate were defined for them; it would be a generic function. However, at present there is no such predicate.

Currently `make-load-form` provides a protocol by which `compile-file` and `load` work together to construct an object in the compiled code that is equivalent to the object in the source code; a different mechanism would have to be added to permit coalescing.

Note that coalescing is possible only because it is forbidden to destructively modify constants (see `quote`).

Objects containing circular or infinitely recursive references may legitimately appear as constants to be compiled. The compiler is required to preserve `eql`-ness of substructures within a file compiled by `compile-file`.

24.2 Debugging Tools Отладочные средства

The utilities described in this section are sufficiently complex and sufficiently dependent on the host environment that their complete definition is beyond the scope of this book. However, they are also sufficiently useful to warrant mention here. It is expected that every implementation will provide some version of these utilities, however clever or however simple.

Коммунальные услуги, описанные в этом разделе достаточно сложны и достаточно зависит от внешней среды, что их полное описание выходит за рамки этой книги. Тем не менее, они также достаточно полезно, чтобы оправдать упомянуть здесь. Ожидается, что каждая реализация предоставить некоторые версии этих программ, однако умный или же просто.

Описанные в этом разделе утилиты достаточно сложны и зависят от внешней среды ОС, что их полное описание выходит за рамки книги. Тем не менее их описание будет полезным. Предполагается, что каждая реализация будет представлять некоторую версию этих утилит.

[Макрос] **trace** {function-name}*
[Макрос] **untrace** {function-name}*

Invoking `trace` with one or more function-names (symbols or lists, whose *car* is `setf`—see section 7.1), causes the functions named to be traced. Henceforth, whenever such a function is invoked, information about the call, the arguments passed, and the eventually returned values, if any, will be printed to the stream that is the value of `*trace-output*`. For example:

(trace fft gcd string-upcase)

If a function call is open-coded (possibly as a result of an `inline` declaration), then such a call may not produce trace output.

Invoking `untrace` with one or more function names will cause those functions not to be traced any more.

Tracing an already traced function, or untracing a function not currently being traced, should produce no harmful effects but may produce a warning message.

Calling `trace` with no argument forms will return a list of functions currently being traced.

Calling `untrace` with no argument forms will cause all currently traced functions to be no longer traced.

The values returned by `trace` and `untrace` when given argument forms are implementation-dependent.

`trace` and `untrace` may also accept additional implementation-dependent argument formats. The format of the trace output is implementation-dependent.

[Макрос] **step** form

This evaluates *form* and returns what *form* returns. However, the user is allowed to interactively "single-step" through the evaluation of *form*, at least through those evaluation steps that are performed interpretively. The nature of the interaction is implementation-dependent. However, implementations are encouraged to respond to the typing of the character `?` by providing help, including a list of commands.

`step` evaluates its argument *form* in the current lexical environment (not simply a null environment), and that calls to `step` may be compiled, in which case an implementation may step through only those parts of the evaluation that are interpreted. (In other words, the *form* itself is unlikely to be stepped, but if executing it happens to invoke interpreted code, then that code may be stepped.)

[Макрос] **time** form

This evaluates *form* and returns what *form* returns. However, as a side effect, various timing data and other information are printed to the stream that is the value of `*trace-output*`. The nature and format of the printed

information is implementation-dependent. However, implementations are encouraged to provide such information as elapsed real time, machine run time, storage management statistics, and so on.

`time` evaluates its argument *form* in the current lexical environment (not simply a null environment), and that calls to `time` may be compiled.

[Function] **describe** *object* **&optional** *stream*

`describe` prints, to the stream information about the *object*. Sometimes it will describe something that it finds inside something else; such recursive descriptions are indented appropriately. For instance, `describe` of a symbol will exhibit the symbol's value, its definition, and each of its properties. `describe` of a floating-point number will exhibit its internal representation in a way that is useful for tracking down round-off errors and the like. The nature and format of the output is implementation-dependent.

`describe` returns no values (that is, it returns what the expression `(values)` returns: zero values).

The output is sent to the specified *stream*, which defaults to the value of `*standard-output*`; the *stream* may also be `nil` (meaning `*standard-output*`) or `t` (meaning `*terminal-io*`).

The behavior of `describe` depends on the generic function `describe-object` (see below).

That `describe` is forbidden to prompt for or require user input when given exactly one argument; It is permitted implementations to extend `describe` to accept keyword arguments that may cause it to prompt for or to require user input.

[Generic function] **describe-object** *object stream*
[Primary method] **describe-object** (*object* standard-object) *stream*

The generic function `describe-object` writes a description of an object to a stream. The function `describe-object` is called by the `describe` function; it should not be called by the user.

Each implementation must provide a method on the class `standard-object` and methods on enough other classes to ensure that there is always an applicable method. Implementations are free to add methods for other classes. Users can write methods for `describe-object` for their own classes if they do not wish to inherit an implementation-supplied method.

The first argument may be any Lisp object. The second argument is a stream; it cannot be `t` or `nil`. The values returned by `describe-object` are unspecified.

Methods on `describe-object` may recursively call `describe`. Indentation, depth limits, and circularity detection are all taken care of automatically, provided that each method handles exactly one level of structure and calls `describe` recursively if there are more structural levels. If this rule is not obeyed, the results are undefined.

In some implementations the *stream* argument passed to a `describe-object` method is not the original stream but is an intermediate stream that implements parts of `describe`. Methods should therefore not depend on the identity of this stream.

Обоснование: This proposal was closely modeled on the CLOS description of `print-object`, which was well thought out and provides a great deal of functionality and implementation freedom. Implementation techniques for `print-object` are applicable to `describe-object`.

The reason for making the return values for `describe-object` unspecified is to avoid forcing users to write `(values)` explicitly in all their methods; `describe` should take care of that.

[Function] **inspect** *object*

`inspect` is an interactive version of `describe`. The nature of the interaction is implementation-dependent, but the purpose of `inspect` is to make it easy to wander through a data structure, examining and modifying parts of it. Implementations are encouraged to respond to the typing of the character `?` by providing help, including a list of commands.

The values returned by `inspect` are implementation-dependent.

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

`room` prints, to the stream in the variable `*standard-output*`, information about the state of internal storage and its management. This might include descriptions of the amount of memory in use and the degree of memory compaction, possibly broken down by internal data type if that is appropriate. The nature and format of the printed information is implementation-dependent. The intent is to provide information that may help a user to tune a program to a particular implementation.

`(room nil)` prints out a minimal amount of information. `(room t)` prints out a maximal amount of information. Simply `(room)` prints out an intermediate amount of information that is likely to be useful.

The argument *x* may also be the keyword `:default`, which has the same effect as passing no argument at all.

[Function] **ed &optional** *x*

If the implementation provides a resident editor, this function should invoke it.

`(ed)` or `(ed nil)` simply enters the editor, leaving you in the same state as the last time you were in the editor.

`(ed` *`pathname`*`)` edits the contents of the file specified by *pathname*. The *pathname* may be an actual pathname or a string.

`ed` accepts logical pathnames (see section 23.1.5).

`(ed` *`symbol`*`)` tries to let you edit the text for the function named *symbol*. The means by which the function text is obtained is implementation-dependent; it might involve searching the file system, or pretty printing resident interpreted code, for example.

Function name may be any function-name (a symbol or a list whose *car* is `setf`—see section 7.1). Thus one may write `(ed '(setf cadr))` to edit the `setf` expansion function for `cadr`.

[Function] **dribble &optional** *pathname*

`(dribble` *`pathname`*`)` may rebind `*standard-input*` and `*standard-output*`, and may take other appropriate action, so as to send a record of the input/output interaction to a file named by *pathname*. The primary purpose of this is to create a readable record of an interactive session.

`(dribble)` terminates the recording of input and output and closes the dribble file.

`dribble` also accepts logical pathnames (see section 23.1.5).

`dribble` is intended primarily for interactive debugging and that its effect cannot be relied upon for use in portable programs.

Different implementations of Common Lisp have used radically different techniques for implementing `dribble`. All are reasonable interpretations of the original specification, and all behave in approximately the same way if

`dribble` is called only from the interactive top level. However, they may have quite different behaviors if `dribble` is called from within compound forms.

Consider two models of the operation of `dribble`. In the "redirecting" model, a call to `dribble` with a pathname argument alters certain global variables such as `*standard-output*`, perhaps by constructing a broadcast stream directed to both the original value of `*standard-output*` and to the dribble file; other streams may be affected as well. A call to `dribble` with no arguments undoes these side effects.

In the "recursive" model, by contrast, a call to `dribble` with a pathname argument creates a new interactive command loop and calls it recursively. This new command loop is just like an ordinary read-eval-print loop except that it also echoes the interaction to the dribble file. A call to `dribble` with no arguments does a `throw` that exits the recursive command loop and returns to the original caller of `dribble` with an argument.

The two models may be distinguished by this test case:

```
(progn (dribble "basketball")
       (print "Larry")
       (dribble)
       (princ "Bird"))
```

If this form is input to the Lisp top level, in either model a newline (provided by the function `print`) and the words `Larry Bird` will be printed to the standard output. The redirecting dribble model will additionally print all but the word `Bird` to a file named `basketball`.

By contrast, the recursive dribble model will enter a recursive command loop and not print anything until `(dribble)` is executed from within the new interactive command loop. At that time the file named `basketball` will be closed, and then execution of the `progn` form will be resumed. A newline and "`Larry` " (note the trailing space) will be printed to the standard output, and then the call `(dribble)` may complain that there is no active dribble file. Once this error is resolved, the word `Bird` may be printed to the standard output.

Here is a slightly different test case:

```
(dribble "baby-food")
```

```
(progn (print "Mashed banana")
       (dribble)
       (princ "and cream of rice"))
```

If this form is input to the Lisp top level, in the redirecting model a newline and the words `Mashed banana and cream of rice` will be printed to the standard output and all but the words `and cream of rice` will be sent to a file named `baby-food`.

The recursive model will direct exactly the same output to the file named `baby-food` but will never print the words `and cream of rice` to the standard output because the call `(dribble)` does not return normally; it throws.

The redirecting model may be intuitively more appealing to some. The recursive model, however, may be more robust; it carefully limits the extent of the dribble operation and disables dribbling if a throw of any kind occurs. The vote by X3J13 was an explicit decision not to decide which model to use. Users are advised to call `dribble` only interactively, at top level.

[Function] **apropos** *string* **&optional** *package*
[Function] **apropos-list** *string* **&optional** *package*

`(apropos` ***string***`)` tries to find all available symbols whose print names contain *string* as a substring. (A symbol may be supplied for the *string*, in which case the print name of the symbol is used.) Whenever `apropos` finds a symbol, it prints out the symbol's name; in addition, information about the function definition and dynamic value of the symbol, if any, is printed. If *package* is specified and not `nil`, then only symbols available in that package are examined; otherwise "all" packages are searched, as if by `do-all-symbols`. Because a symbol may be available by way of more than one inheritance path, `apropos` may print information about the same symbol more than once. The information is printed to the stream that is the value of `*standard-output*`. `apropos` returns no values (that is, it returns what the expression `(values)` returns: zero values).

`apropos-list` performs the same search that `apropos` does but prints nothing. It returns a list of the symbols whose print names contain *string* as a substring.

24.3 Environment Inquiries Справка о среде

Environment inquiry functions provide information about the environment in which a Common Lisp program is being executed. They are described here in two categories: first, those dealing with determination and measurement of time, and second, all the others, most of which deal with identification of the computer hardware and software.

Справочные функции представляют информацию о среде, в которой исполняется Common Lisp'овая программа. Функции разделены на две категории: первые для работы со временем, и остальные для получения имен, версий, типов программ и оборудования.

24.3.1 Time Functions

Time is represented in three different ways in Common Lisp: Decoded Time, Universal Time, and Internal Time. The first two representations are used primarily to represent calendar time and are precise only to one second. Internal Time is used primarily to represent measurements of computer time (such as run time) and is precise to some implementation-dependent fraction of a second, as specified by `internal-time-units-per-second`. Decoded Time format is used only for absolute time indications. Universal Time and Internal Time formats are used for both absolute and relative times.

Decoded Time format represents calendar time as a number of components:

- *Second*: an integer between 0 and 59, inclusive.
- *Minute*: an integer between 0 and 59, inclusive.
- *Hour*: an integer between 0 and 23, inclusive.
- *Date*: an integer between 1 and 31, inclusive (the upper limit actually depends on the month and year, of course).
- *Month*: an integer between 1 and 12, inclusive; 1 means January, 12 means December.
- *Year*: an integer indicating the year A.D. However, if this integer is between 0 and 99, the "obvious" year is used; more precisely, that year

is assumed that is equal to the integer modulo 100 and within fifty years of the current year (inclusive backwards and exclusive forwards). Thus, in the year 1978, year 28 is 1928 but year 27 is 2027. (Functions that return time in this format always return a full year number.)

- *Day-of-week*: an integer between 0 and 6, inclusive; 0 means Monday, 1 means Tuesday, and so on; 6 means Sunday.
- *Daylight-saving-time-p*: a flag that, if not `nil`, indicates that daylight saving time is in effect.
- *Time-zone*: an integer specified as the number of hours west of GMT (Greenwich Mean Time). For example, in Massachusetts the time zone is 5, and in California it is 8. Any adjustment for daylight saving time is separate from this.

Time zone part of Decoded Time need not be an integer, but may be any rational number (either an integer or a ratio) in the range -24 to 24 (inclusive on both ends) that is an integral multiple of `1/3600`.

Обоснование: For all possible time designations to be accommodated, it is necessary to allow the time zone to be non-integral, for some places in the world have time standards offset from Greenwich Mean Time by a non-integral number of hours.

There appears to be no user demand for floating-point time zones. Since such zones would introduce inexact arithmetic, X3J13 did not consider adding them at this time.

This specification does require time zones to be represented as integral multiples of 1 second (rather than 1 hour). This prevents problems that could otherwise occur in converting Decoded Time to Universal Time.

Universal Time represents time as a single non-negative integer. For relative time purposes, this is a number of seconds. For absolute time, this is the number of seconds since midnight, January 1, 1900 GMT. Thus the time 1 is 00:00:01 (that is, 12:00:01 A.M.) on January 1, 1900 GMT. Similarly, the time 2398291201 corresponds to time 00:00:01 on January 1, 1976 GMT. Recall that the year 1900 was *not* a leap year; for the purposes of Common Lisp, a year is a leap year if and only if its number is divisible by 4, except that years divisible by 100 are *not* leap years, except that years divisible

by 400 *are* leap years. Therefore the year 2000 will be a leap year. (Note that the "leap seconds" that are sporadically inserted by the world's official timekeepers as an additional correction are ignored; Common Lisp assumes that every day is exactly 86400 seconds long.) Universal Time format is used as a standard time representation within the ARPANET; see reference [22]. Because the Common Lisp Universal Time representation uses only non-negative integers, times before the base time of midnight, January 1, 1900 GMT cannot be processed by Common Lisp.

Internal Time also represents time as a single integer, but in terms of an implementation-dependent unit. Relative time is measured as a number of these units. Absolute time is relative to an arbitrary time base, typically the time at which the system began running.

[Function] **get-decoded-time**

The current time is returned in Decoded Time format. Nine values are returned: *second*, *minute*, *hour*, *date*, *month*, *year*, *day-of-week*, *daylight-saving-time-p*, and *time-zone*.

[Function] **get-universal-time**

The current time of day is returned as a single integer in Universal Time format.

Функция возвращает текущее время всемирное время в

[Function] **decode-universal-time** *universal-time* **&optional** *time-zone*

The time specified by *universal-time* in Universal Time format is converted to Decoded Time format. Nine values are returned: *second*, *minute*, *hour*, *date*, *month*, *year*, *day-of-week*, *daylight-saving-time-p*, and *time-zone*.

The *time-zone* argument defaults to the current time zone.

`decode-universal-time`, like `encode-universal-time`, ignores daylight saving time information if a *time-zone* is explicitly specified; in this case the returned *daylight-saving-time-p* value will necessarily be `nil` even if daylight saving time happens to be in effect in that time zone at the specified time.

[Function] **encode-universal-time** *second minute hour date month year* **&optional** *time-zone*

The time specified by the given components of Decoded Time format is encoded into Universal Time format and returned. If you do not specify

time-zone, it defaults to the current time zone adjusted for daylight saving time. If you provide *time-zone* explicitly, no adjustment for daylight saving time is performed.

Функция преобразует время, заданное компонентами формата декодированного времени, в формат всемирного времени. Если вы не укажете часовой пояс *time-zone*, то он будет по-умолчанию равен текущему часовому поясу с учетом перехода на летнее время. Если вы укажете явно часовой пояс *time-zon*, учет летнего времени производиться не будет.

[Constant] **internal-time-units-per-second**

This value is an integer, the implementation-dependent number of internal time units in a second. (The internal time unit must be chosen so that one second is an integral multiple of it.)

Значение константы является целым числом и зависит от того, сколько единиц внутреннего времени в секунде для данной реализации Common Lisp'a. (Единица внутреннего времени должна быть выбрана так, чтобы в секунде помещалось целое число таких единиц.)

Обоснование: The reason for allowing the internal time units to be implementation-dependent is so that `get-internal-run-time` and `get-internal-real-time` can execute with minimum overhead. The idea is that it should be very likely that a fixnum will suffice as the returned value from these functions. This probability can be tuned to the implementation by trading off the speed of the machine against the word size. Any particular unit will be inappropriate for some implementations: a microsecond is too long for a very fast machine, while a much smaller unit would force many implementations to return bignums for most calls to `get-internal-time`, rendering that function less useful for accurate timing measurements.

[Function] **get-internal-run-time**

The current run time is returned as a single integer in Internal Time format. The precise meaning of this quantity is implementation-dependent; it may measure real time, run time, CPU cycles, or some other quantity. The intent is that the difference between the values of two calls to this function be the amount of time between the two calls during which computational effort was expended on behalf of the executing program.

Функция возвращает целое число в формате Внутреннего Времени. Точное значение этой величины зависит от реализации и может измерятся в реальном времени, времени выполнения, циклов центрального процессора, или в чем-либо другом. Суть в том, что разница между значениями двух вызовов будет количеством времени, потраченным на исполнение программы между этими вызовами.

[Function] **get-internal-real-time**

The current time is returned as a single integer in Internal Time format. This time is relative to an arbitrary time base, but the difference between the values of two calls to this function will be the amount of elapsed real time between the two calls, measured in the units defined by `internal-time-units-per-second`.

Функция возвращает целое число в формате Внутреннего Времени. Данное время относительно некоторого базового значения, но разница между значениями двух вызовов этой функции будет количеством прошедшего (между этими вызовами) реального времени, выраженным в `internal-time-units-per-second`.

[Function] **sleep** *seconds*

`(sleep` ***n***`)` causes execution to cease and become dormant for approximately n seconds of real time, whereupon execution is resumed. The argument may be any non-negative non-complex number. `sleep` returns `nil`.

`(sleep` ***n***`)` приостанавливает исполнение примерно на n секунд в реальном времени. Аргумент может быть любым неотрицательным некомплексным числом. `sleep` возвращает `nil`.

24.3.2 Other Environment Inquiries Справочные функции о среде

For any of the following functions, if no appropriate and relevant result can be produced, `nil` is returned instead of a string.

Любая из этих функций вместо строки может возвращать результат `nil`, если подходящей информации нет.

Обоснование: These inquiry facilities are functions rather than variables against the possibility that a Common Lisp process might migrate from machine to ma-

chine. This need not happen in a distributed environment; consider, for example, dumping a core image file containing a compiler and then shipping it to another site.

Обоснование: Эти справочные данные возвращаются функциями, а не хранятся переменных, так как процесс Common Lisp'a может мигрировать между компьютерами (машинами). Это необязтально случается в распределенной среде. Например может служить сохранение образа содержащего компилятор и затем восстановление данного образа на другой машине.

[Function] **lisp-implementation-type**

A string is returned that identifies the generic name of the particular Common Lisp implementation. Examples: `"Spice LISP"`, `"Zetalisp"`, `"SBCL"`.

Функция возвращают имя текущей реализации Common Lisp'a. Примеры: `"Spice LISP"`, `"Zetalisp"`, `"SBCL"`.

[Function] **lisp-implementation-version**

A string is returned that identifies the version of the particular Common Lisp implementation; this information should be of use to maintainers of the implementation. Examples: `"1192"`, `"53.7 with complex numbers"`, `"1746.9A, NEWIO 53, ETHER 5.3"`.

Функция возвращает версию текущей реализации Common Lisp'a. Эта информация должно быть использована сопровождающими реализацию. Примеры: `"1192"`, `"53.7 with complex numbers"`, `"1746.9A, NEWIO 53, ETHER 5.3"`.

[Function] **machine-type**

A string is returned that identifies the generic name of the computer hardware on which Common Lisp is running. Examples: `"IMLAC"`, `"DEC PDP-10"`, `"DEC VAX-11/780"`, `"X86-64"`.

Функция возвращает имя типа аппаратного обеспечения, на котором запущен Common Lisp. Примеры: `"IMLAC"`, `"DEC PDP-10"`, `"DEC VAX-11/780"`, `"X86-64"`.

[Function] **machine-version**

A string is returned that identifies the version of the computer hardware on which Common Lisp is running. Example: `"KL10, microcode 9"`, `"AMD Athlon(tm) 64 X2 Dual Core Processor 3600+"`.

Функция возвращает версию аппаратного обеспечения, на котором запущен Common Lisp. Примеры: `"KL10, microcode 9"`, `"AMD Athlon(tm) 64 X2 Dual Core Processor 3600+"`.

[Function] **machine-instance**

A string is returned that identifies the particular instance of the computer hardware on which Common Lisp is running; this might be a local nickname, for example, or a serial number. Examples: `"MIT-MC"`, `"CMU GP-VAX"`.

Функция возвращает имя компьютера, на котором запущена реализация Common Lisp'а. Примеры: `"MIT-MC"`, `"CMU GP-VAX"`.

[Function] **software-type**

A string is returned that identifies the generic name of any relevant supporting software. Examples: `"Spice"`, `"TOPS-20"`, `"ITS"`, `Linux`.

Функция возвращает имя типа текущей операционной системы. Примеры: `"Spice"`, `"TOPS-20"`, `"ITS"`, `Linux`.

[Function] **software-version**

A string is returned that identifies the version of any relevant supporting software; this information should be of use to maintainer of the implementation.

Функция возвращает версию текущей операционной системы. Данная информация должно использоваться сопровождающими ОС. Пример для ArchLinux'а: `"3.2.13-1-ARCH"`.

[Function] **short-site-name**
[Function] **long-site-name**

A string is returned that identifies the physical location of the computer hardware. Examples of short names: `"MIT AI Lab"`, `"CMU-CSD"`. Examples of long names:

"MIT Artificial Intelligence Laboratory"
"Massachusetts Institute of Technology
Artificial Intelligence Laboratory"
"Carnegie-Mellon University Computer Science Department"

Функции возвращают строки, обозначающие физическое расположение аппаратной части компьютера. Примеры для кратких имен: `"MIT AI Lab"`, `"CMU-CSD"`. Примеры для длинных имен:

"MIT Artificial Intelligence Laboratory"
"Massachusetts Institute of Technology
Artificial Intelligence Laboratory"
"Carnegie-Mellon University Computer Science Department"

See also `user-homedir-pathname`.
Смотрите также `user-homedir-pathname`.

[Variable] ***features***

The value of the variable `*features*` should be a list of symbols that name "features" provided by the implementation. Most such names will be implementation-specific; typically a name for the implementation will be included.

Значение переменной `*features` должно быть списком символов, которые указывают на имена «возможностей» данной реализации. Большинство этих имен специфичны.

The value of this variable is used by the `#+` and `#-` reader syntax.

Значение этой переменной используется с помощью синтаксических конструкций считывателя: `#+` и `#-`.

Feature names used with `#+` and `#-` are read in the `keyword` package unless an explicit prefix designating some other package appears. The standard feature name `ieee-floating-point` is therefore actually the keyword `:ieee-floating-point`, though one need not write the colon when using it with `#+` or `#-`; thus `#+ieee-floating-point` and `#+:ieee-floating-point` mean the same thing.

По-умолчанию символы для имен «возможностей», использованные в конструкциях `#+` и `#-`, ищутся в пакете `keyword`. Таким образом

`#+ieee-floating-point` и `#+:ieee-floating-point` означают одно и то же.

24.4 Identity Function Функция идентичности (identity)

This function is occasionally useful as an argument to other functions that require functions as arguments. (Got that?)

Эта функция иногда бывает полезна для использования в качестве аргумента других функций, которые требуют функции в качестве аргументов. (Понятно?)

[Function] **identity** *object*

The *object* is returned as the value of `identity`.

The `identity` function is the default value for the `:key` argument to many sequence functions (see chapter 14).

Table 12.1 illustrates the behavior in the complex plane of the `identity` function regarded as a function of a complex numerical argument.

Many other constructs in Common Lisp have the behavior of `identity` when given a single argument. For example, one might well use `values` in place of `identity`. However, writing `values` of a single argument conventionally indicates that the argument form might deliver multiple values and that the intent is to pass on only the first of those values.

Результатом функции является переданный объект *object*.

Функция `identity` используется по-умолчанию для аргумента `:key` для большинства функций для последовательностей (смотрите главу 14).

Поведение функции `identity` для комплексного числа проиллюстрировано в таблице 12.1.

Множество других Common Lisp'овых конструкций с одним аргументом ведут себя так же как и *identity*. Например, можно использовать `values` вместо `identity`. Однако, запись `values` с одним аргументом означает, что форма аргумента возвращает несколько значений, но необходимо вернуть только одно из них.

[Function] **constantly** *object*

Returns a function that will always return the *object*. The returned function takes any number of arguments.

[Функция] **constantly** *object*

Функция возвращает другую функцию, которая всегда возвращает объект *object*. Возвращенная функция принимает любое количество аргументов.

[Макрос] **lambda** lambda-list [[{declaration}* | [doc-string]] {form}*

A dubious shortcut for `(function (lambda ...))` or `#'(lambda ...)`.

Двусмысленное сокращение для `(function (lambda ...))` или `#'(lambda ...)`.

Глава 25

Loop

Author: Jon L White

preface:X3J13 voted in January 1989 to adopt an extended definition of the `loop` macro as a part of the forthcoming draft Common Lisp standard. This chapter presents the bulk of the Common Lisp Loop Facility proposal, written by Jon L White. I have edited it only very lightly to conform to the overall style of this book and have inserted a small number of bracketed remarks, identified by the initials GLS. (See the Acknowledgments to this second edition for acknowledgments to others who contributed to the Loop Facility proposal.)

Guy L. Steele Jr.

25.1 Introduction

A *loop* is a series of expressions that are executed one or more times, a process known as *iteration*. The *Loop Facility* defines a variety of useful methods, indicated by *loop keywords*, to iterate and to accumulate values in a loop.

Loop keywords are not true Common Lisp keywords; they are symbols that are recognized by the Loop Facility and that provide such capabilities as controlling the direction of iteration, accumulating values inside the body of a loop, and evaluating expressions that precede or follow the loop body. If you do not use any loop keywords, the Loop Facility simply executes the loop body repeatedly.

25.2 How the Loop Facility Works

The driving element of the Loop Facility is the `loop` macro. When Lisp encounters a `loop` macro call form, it invokes the Loop Facility and passes to it the loop clauses as a list of unevaluated forms, as with any macro. The loop clauses contain Common Lisp forms and loop keywords. The loop keywords are recognized by their symbol name, regardless of the packages that contain them. The `loop` macro translates the given form into Common Lisp code and returns the expanded form.

The expanded loop form is one or more lambda-expressions for the local binding of loop variables and a block and a tagbody that express a looping control structure. The variables established in the loop construct are bound as if by using `let` or `lambda`. Implementations can interleave the setting of initial values with the bindings. However, the assignment of the initial values is always calculated in the order specified by the user. A variable is thus sometimes bound to a harmless value of the correct data type, and then later in the prologue it is set to the true initial value by using `setq`.

The expanded form consists of three basic parts in the tagbody:

- The *loop prologue* contains forms that are executed before iteration begins, such as initial settings of loop variables and possibly an initial termination test.

- The *loop body* contains those forms that are executed during iteration, including application-specific calculations, termination tests, and variable stepping. *Stepping* is the process of assigning a variable the next item in a series of items.

- The *loop epilogue* contains forms that are executed after iteration terminates, such as code to return values from the loop.

Expansion of the `loop` macro produces an implicit block (named `nil`). Thus, the Common Lisp macro `return` and the special operator `return-from` can be used to return values from a loop or to exit a loop.

Within the executable parts of loop clauses and around the entire loop form, you can still bind variables by using the Common Lisp special operator `let`.

25.3 Parsing Loop Clauses

The syntactic parts of a loop construct are called *clauses*; the scope of each clause is determined by the top-level parsing of that clause's keyword. The following example shows a loop construct with six clauses:

```
(loop for i from 1 to (compute-top-value)        ;First clause
      while (not (unacceptable i))               ;Second clause
      collect (square i)                         ;Third clause
      do (format t "Working on ~D now" i)        ;Fourth clause
      when (evenp i)                             ;Fifth clause
        do (format t "~D is a non-odd number" i)
      finally (format t "About to exit!"))       ;Sixth clause
```

Each loop keyword introduces either a compound loop clause or a simple loop clause that can consist of a loop keyword followed by a single Lisp form. The number of forms in a clause is determined by the loop keyword that begins the clause and by the auxiliary keywords in the clause. The keywords `do`, `initially`, and `finally` are the only loop keywords that can take any number of Lisp forms and group them as if in a single `progn` form.

Loop clauses can contain auxiliary keywords, which are sometimes called *prepositions*. For example, the first clause in the preceding code includes the prepositions `from` and `to`, which mark the value from which stepping begins and the value at which stepping ends.

25.3.1 Order of Execution

With the exceptions listed below, clauses are executed in the loop body in the order in which they appear in the source. Execution is repeated until a clause terminates the loop or until a Common Lisp `return`, `go`, or `throw` form is encountered. The following actions are exceptions to the linear order of execution:

- All variables are initialized first, regardless of where the establishing clauses appear in the source. The order of initialization follows the order of these clauses.

- The code for any `initially` clauses is collected into one `progn` in the order in which the clauses appear in the source. The collected code is executed once in the loop prologue after any implicit variable initializations.

- The code for any `finally` clauses is collected into one `progn` in the order in which the clauses appear in the source. The collected code is executed once in the loop epilogue before any implicit values from the accumulation clauses are returned. Explicit returns anywhere in the source, however, will exit the loop without executing the epilogue code.

- A `with` clause introduces a variable binding and an optional initial value. The initial values are calculated in the order in which the `with` clauses occur.

- Iteration control clauses implicitly perform the following actions:

  – initializing variables

  – stepping variables, generally between each execution of the loop body

  – performing termination tests, generally just before the execution of the loop body

25.3.2 Kinds of Loop Clauses

Loop clauses fall into one of the following categories:

- variable initialization and stepping

  – The `for` and `as` constructs provide iteration control clauses that establish a variable to be initialized. You can combine `for` and `as` clauses with the loop keyword `and` to get parallel initialization and stepping.

  - The `with` construct is similar to a single `let` clause. You can combine `with` clauses using `and` to get parallel initialization.
  - The `repeat` construct causes iteration to terminate after a specified number of times. It uses an internal variable to keep track of the number of iterations.

  You can specify data types for loop variables (see section 25.12.1). It is an error to bind the same variable twice in any variable-binding clause of a single loop expression. Such variables include local variables, iteration control variables, and variables found by destructuring.

- value accumulation

  - The `collect` construct takes one form in its clause and adds the value of that form to the end of a list of values. By default, the list of values is returned when the loop finishes.
  - The `append` construct takes one form in its clause and appends the value of that form to the end of a list of values. By default, the list of values is returned when the loop finishes.
  - The `nconc` construct is similar to `append`, but its list values are concatenated as if by the Common Lisp function `nconc`. By default, the list of values is returned when the loop finishes.
  - The `sum` construct takes one form in its clause that must evaluate to a number and adds that number into a running total. By default, the cumulative sum is returned when the loop finishes.
  - The `count` construct takes one form in its clause and counts the number of times that the form evaluates to a non-`nil` value. By default, the count is returned when the loop finishes.
  - The `minimize` construct takes one form in its clause and determines the minimum value obtained by evaluating that form. By default, the minimum value is returned when the loop finishes.
  - The `maximize` construct takes one form in its clause and determines the maximum value obtained by evaluating that form. By default, the maximum value is returned when the loop finishes.

- termination conditions

  - The `loop-finish` Lisp macro terminates iteration and returns any accumulated result. If specified, any `finally` clauses are evaluated.
  - The `for` and `as` constructs provide a termination test that is determined by the iteration control clause.
  - The `repeat` construct causes termination after a specified number of iterations.
  - The `while` construct takes one form, a condition, and terminates the iteration if the condition evaluates to `nil`. A `while` clause is equivalent to the expression `(if (not` *`condition`*`) (loop-finish))`.
  - The `until` construct is the inverse of `while`; it terminates the iteration if the condition evaluates to any non-`nil` value. An `until` clause is equivalent to the expression `(if` *`condition`* `(loop-finish))`.
  - The `always` construct takes one form and terminates the loop if the form ever evaluates to `nil`; in this case, it returns `nil`. Otherwise, it provides a default return value of `t`.
  - The `never` construct takes one form and terminates the loop if the form ever evaluates to non-`nil`; in this case, it returns `nil`. Otherwise, it provides a default return value of `t`.
  - The `thereis` construct takes one form and terminates the loop if the form ever evaluates to non-`nil`; in this case, it returns that value.

- unconditional execution

  - The `do` construct simply evaluates all forms in its clause.
  - The `return` construct takes one form and returns its value. It is equivalent to the clause `do (return` *`value`*`)`.

- conditional execution
  - The `if` construct takes one form as a predicate and a clause that is executed when the predicate is true. The clause can be a value accumulation, unconditional, or another conditional clause; it can also be any combination of such clauses connected by the loop keyword `and`.
  - The `when` construct is a synonym for `if`.
  - The `unless` construct is similar to `when` except that it complements the predicate; it executes the following clause if the predicate is false.
  - The `else` construct provides an optional component of `if`, `when`, and `unless` clauses that is executed when the predicate is false. The component is one of the clauses described under `if`.
  - The `end` construct provides an optional component to mark the end of a conditional clause.
- miscellaneous operations
  - The `named` construct assigns a name to a loop construct.
  - The `initially` construct causes its forms to be evaluated in the loop prologue, which precedes all loop code except for initial settings specified by the constructs `with`, `for`, or `as`.
  - The `finally` construct causes its forms to be evaluated in the loop epilogue after normal iteration terminates. An unconditional clause can also follow the loop keyword `finally`.

25.3.3 Loop Syntax

The following syntax description provides an overview of the syntax for loop clauses. Detailed syntax descriptions of individual clauses appear in sections 25.6 through 25.12. A loop consists of the following types of clauses:

initial-final ::= *initially* | *finally*
variables ::= *with* | *initial-final* | *for-as* | *repeat*
main ::= *unconditional* | *accumulation* | *conditional* | *termination* | *initial-final*
loop ::= `(loop [`*named* `name] {}* variables {}* main)`

Note that a loop must have at least one clause; however, for backward compatibility, the following format is also supported:

(loop {tag / expr}*)

where *expr* is any Common Lisp expression that can be evaluated, and *tag* is any symbol not identifiable as a loop keyword. Such a format is roughly equivalent to the following one:

(loop do {tag / expr}*)

A loop prologue consists of any automatic variable initializations prescribed by the *variable* clauses, along with any *initially* clauses in the order they appear in the source.

A loop epilogue consists of *finally* clauses, if any, along with any implicit return value from an *accumulation* clause or an *end-test* clause.

25.4 User Extensibility

There is currently no specified portable method for users to add extensions to the Loop Facility. The names `defloop` and `define-loop-method` have been suggested as candidates for such a method.

25.5 Loop Constructs

The remaining sections of this chapter describe the constructs that the Loop Facility provides. The descriptions are organized according to the functionality of the constructs. Each section begins with a general discussion of a

particular operation; it then presents the constructs that perform the operation.

- Section 25.6, "Iteration Control," describes iteration control clauses that allow directed loop iteration.
- Section 25.7, "End-Test Control," describes clauses that stop iteration by providing a conditional expression that can be tested after each execution of the loop body.
- Section 25.8, "Value Accumulation," describes constructs that accumulate values during iteration and return them from a loop. This section also discusses ways in which accumulation clauses can be combined within the Loop Facility.
- Section 25.9, "Variable Initializations," describes the `with` construct, which provides local variables for use within the loop body, and other constructs that provide local variables.
- Section 25.10, "Conditional Execution," describes how to execute loop clauses conditionally.
- Section 25.11, "Unconditional Execution," describes the `do` and `return` constructs. It also describes constructs that are used in the loop prologue and loop epilogue.
- Section 25.12, "Miscellaneous Features," discusses loop data types and destructuring. It also presents constructs for naming a loop and for specifying initial and final actions.

25.6 Iteration Control

Iteration control clauses allow you to direct loop iteration. The loop keywords `as`, `for`, and `repeat` designate iteration control clauses.

Iteration control clauses differ with respect to the specification of termination conditions and the initialization and stepping of loop variables. Iteration clauses by themselves do not cause the Loop Facility to return values, but

they can be used in conjunction with value-accumulation clauses to return values (see section 25.8).

All variables are initialized in the loop prologue. The scope of the variable binding is *lexical* unless it is proclaimed special; thus, the variable can be accessed only by expressions that lie textually within the loop. Stepping assignments are made in the loop body before any other expressions are evaluated in the body.

The variable argument in iteration control clauses can be a *destructuring list*. A destructuring list is a tree whose non-null atoms are symbols that can be assigned a value (see section 25.12.2).

The iteration control clauses `for`, `as`, and `repeat` must precede any other loop clauses except `initially`, `with`, and `named`, since they establish variable bindings. When iteration control clauses are used in a loop, termination tests in the loop body are evaluated before any other loop body code is executed.

If you use multiple iteration clauses to control iteration, variable initialization and stepping occur sequentially by default. You can use the `and` construct to connect two or more iteration clauses when sequential binding and stepping are not necessary. The iteration behavior of clauses joined by `and` is analogous to the behavior of the Common Lisp macro `do` relative to `do*`.

[X3J13 voted in March 1989 to correct a minor inconsistency in the original syntactic specification for `loop`. Only `for` and `as` clauses (not `repeat` clauses) may be joined by the `and` construct. The precise syntax is as follows.

for-as ::= {**for** / **as**} *for-as-subclause* {**and** for-as-subclause}*
for-as-subclause ::= *for-as-arithmetic* | *for-as-in-list*
| *for-as-on-list* | *for-as-equals-then*
| *for-as-across* | *for-as-hash* | *for-as-package*
for-as-arithmetic ::= *var* [*type-spec*] [{**from** / **downfrom** / **upfrom**} *expr1*]
[{**to** / **downto** / **upto** / **below** / **above**} *expr2*]
[**by** *expr3*]
for-as-in-list ::= *var* [*type-spec*] `in` *expr1* [**by** *step-fun*]
for-as-on-list ::= *var* [*type-spec*] `on` *expr1* [**by** *step-fun*]
for-as-equals-then ::= *var* [*type-spec*] `=` *expr1* [**then** *step-fun*]
for-as-across ::= *var* [*type-spec*] `across` *vector*

for-as-hash ::= *var* [*type-spec*] `being` {*each* / *the*}
{*hash-key* / *hash-keys* / *hash-value* / *hash-values*}
{*in* / *of*} *hash-table*
[*using* (*{hash-value* / *hash-key}* *other-var*)]
for-as-package ::= *var* [*type-spec*] `being` {*each* / *the*}
for-as-package-keyword
{*in* / *of*} *package*
for-as-package-keyword ::= `symbol` | `present-symbol` | `external-symbol`
| `symbols` | `present-symbols` | `external-symbols`

This correction made `for` and `as` clauses syntactically similar to `with` clauses. I have changed all examples in this chapter to reflect the corrected syntax.—GLS]

In the following example, the variable `x` is stepped before `y` is stepped; thus, the value of `y` reflects the updated value of `x`:

```
(loop for x from 1 to 9
      for y = nil then x
      collect (list x y))
  ⇒ ((1 NIL) (2 2) (3 3) (4 4) (5 5) (6 6) (7 7) (8 8) (9 9))
```

In the following example, `x` and `y` are stepped in parallel:

```
(loop for x from 1 to 9
      and y = nil then x
      collect (list x y))
  ⇒ ((1 NIL) (2 1) (3 2) (4 3) (5 4) (6 5) (7 6) (8 7) (9 8))
```

The `for` and `as` clauses iterate by using one or more local loop variables that are initialized to some value and that can be modified or stepped after each iteration. For these clauses, iteration terminates when a local variable reaches some specified value or when some other loop clause terminates iteration. At each iteration, variables can be stepped by an increment or a decrement or can be assigned a new value by the evaluation of an expression. Destructuring can be used to assign initial values to variables during iteration.

The `for` and `as` keywords are synonyms and may be used interchangeably. There are seven syntactic representations for these constructs. In each

syntactic description, the data type of *var* can be specified by the optional *type-spec* argument. If *var* is a destructuring list, the data type specified by the *type-spec* argument must appropriately match the elements of the list (see sections 25.12.1 and 25.12.2).

[Выражение цикла] **for** var [type-spec] [{`from` | `downfrom` | `upfrom`} expr1]
[{`to` | `downto` | `upto` | `below` | `above`} expr2]
[`by` expr3]
[Выражение цикла] **as** var [type-spec] [{`from` | `downfrom` | `upfrom`} expr1]
[{`to` | `downto` | `upto` | `below` | `above`} expr2]
[`by` expr3]

[This is the first of seven `for`/`as` syntaxes.—GLS]

The `for` or `as` construct iterates from the value specified by *expr1* to the value specified by *expr2* in increments or decrements denoted by *expr3*. Each expression is evaluated only once and must evaluate to a number.

The variable *var* is bound to the value of *expr1* in the first iteration and is stepped by the value of *expr3* in each succeeding iteration, or by 1 if *expr3* is not provided.

The following loop keywords serve as valid prepositions within this syntax.

`from` The loop keyword `from` marks the value from which stepping begins, as specified by *expr1*. Stepping is incremental by default. For decremental stepping, use `above` or `downto` with *expr2*. For incremental stepping, the default `from` value is `0`.

`downfrom`, `upfrom` The loop keyword `downfrom` indicates that the variable *var* is decreased in decrements specified by *expr3*; the loop keyword `upfrom` indicates that *var* is increased in increments specified by *expr3*.

`to` The loop keyword `to` marks the end value for stepping specified in *expr2*. Stepping is incremental by default. For decremental stepping, use `downto`, `downfrom`, or `above` with *expr2*.

`downto`, `upto` The loop keyword `downto` allows iteration to proceed from a larger number to a smaller number by the decrement *expr3*. The loop keyword `upto` allows iteration to proceed from a smaller number to a larger number by the increment *expr3*. Since there is no default for *expr1* in decremental stepping, you must supply a value with `downto`.

`below, above` The loop keywords `below` and `above` are analogous to `upto` and `downto`, respectively. These keywords stop iteration just before the value of the variable *var* reaches the value specified by *expr2*; the end value of *expr2* is not included. Since there is no default for *expr1* in decremental stepping, you must supply a value with `above`.

`by` The loop keyword `by` marks the increment or decrement specified by *expr3*. The value of *expr3* can be any positive number. The default value is `1`.

At least one of these prepositions must be used with this syntax.

In an iteration control clause, the `for` or `as` construct causes termination when the specified limit is reached. That is, iteration continues until the value *var* is stepped to the exclusive or inclusive limit specified by *expr2*. The range is *exclusive* if *expr3* increases or decreases *var* to the value of *expr2* without reaching that value; the loop keywords `below` and `above` provide exclusive limits. An *inclusive* limit allows *var* to attain the value of *expr2*; `to`, `downto`, and `upto` provide inclusive limits.

A common convention is to use `for` to introduce new iterations and `as` to introduce iterations that depend on a previous iteration specification. [However, `loop` does not enforce this convention, and some of the examples below violate it. *De gustibus non disputandum est.*—GLS]

Examples:

```
;;; Print some numbers.
(loop as i from 1 to 5
      do (print i))                              ;Prints 5 lines
1
2
3
4
5
  ⇒ NIL

;;; Print every third number.
(loop for i from 10 downto 1 by 3
      do (print i))                              ;Prints 4 lines
```

```
10
7
4
1
  ⇒ NIL
```

```
;;; Step incrementally from the default starting value.
(loop as i below 5
     do (print i))                                    ;Prints 5 lines
0
1
2
3
4
  ⇒ NIL
```

[Выражение цикла] **for** var [type-spec] `in` expr1 [`by` step-fun]
[Выражение цикла] **as** var [type-spec] `in` expr1 [`by` step-fun]

[This is the second of seven `for`/`as` syntaxes.—GLS]

This construct iterates over the contents of a list. It checks for the end of the list as if using the Common Lisp function `endp`. The variable *var* is bound to the successive elements of the list *expr1* before each iteration. At the end of each iteration, the function *step-fun* is called on the list and is expected to produce a successor list; the default value for *step-fun* is the `cdr` function.

The `for` or `as` construct causes termination when the end of the list is reached. The loop keywords `in` and `by` serve as valid prepositions in this syntax.

Examples:

```
;;; Print every item in a list.
(loop for item in '(1 2 3 4 5) do (print item))       ;Prints 5 lines
```

```
1
2
3
4
5
  ⇒ NIL

;;; Print every other item in a list.
(loop for item in '(1 2 3 4 5) by #'cddr
     do (print item))                                    ;Prints 3 lines
1
3
5
  ⇒ NIL



;;; Destructure items of a list, and sum the x values
;;; using fixnum arithmetic.
(loop for (item . x) (t . fixnum)
        in '((A . 1) (B . 2) (C . 3))
     unless (eq item 'B) sum x)
  ⇒ 4
```

[Выражение цикла] **for** var [type-spec] on expr1 [by step-fun]
[Выражение цикла] **as** var [type-spec] on expr1 [by step-fun]

[This is the third of seven `for`/`as` syntaxes.—GLS]

This construct iterates over the contents of a list. It checks for the end of the list as if using the Common Lisp function `endp`. The variable *var* is bound to the successive tails of the list *expr1*. At the end of each iteration, the function *step-fun* is called on the list and is expected to produce a successor list; the default value for *step-fun* is the `cdr` function.

The loop keywords `on` and `by` serve as valid prepositions in this syntax. The `for` or `as` construct causes termination when the end of the list is reached.

Examples:

```
;;; Collect successive tails of a list.
(loop for sublist on '(a b c d)
      collect sublist)
  ⇒ ((A B C D) (B C D) (C D) (D))

;;; Print a list by using destructuring with the loop keyword ON.
(loop for (item) on '(1 2 3)
      do (print item))                                  ;Prints 3 lines
1
2
3
  ⇒ NIL

;;; Print items in a list without using destructuring.
(loop for item in '(1 2 3)
      do (print item))                                  ;Prints 3 lines
1
2
3
  ⇒ NIL
```

[Выражение цикла] **for** var [type-spec] = expr1 [`then` expr2]
[Выражение цикла] **as** var [type-spec] = expr1 [`then` expr2]

[This is the fourth of seven `for`/`as` syntaxes.—GLS]

This construct initializes the variable *var* by setting it to the result of evaluating *expr1* on the first iteration, then setting it to the result of evaluating *expr2* on the second and subsequent iterations. If *expr2* is omitted, the construct uses *expr1* on the second and subsequent iterations. When *expr2* is omitted, the expanded code shows the following optimization:

```
;;; Sample original code:
(loop for x = expr1 then expr2 do (print x))
```

```
;;; The usual expansion:
(tagbody
     (setq x expr1)
  tag (print x)
     (setq x expr2)
     (go tag))
```

```
;;; The optimized expansion:
(tagbody
  tag (setq x expr1)
     (print x)
     (go tag))
```

The loop keywords `=` and `then` serve as valid prepositions in this syntax. This construct does not provide any termination conditions.

Example:

```
;;; Collect some numbers.
(loop for item = 1 then (+ item 10)
     repeat 5
     collect item)
  ⇒ (1 11 21 31 41)
```

[Выражение цикла] **for** var [type-spec] across vector
[Выражение цикла] **as** var [type-spec] across vector

[This is the fifth of seven `for`/`as` syntaxes.—GLS]

This construct binds the variable *var* to the value of each element in the array *vector*.

The loop keyword `across` marks the array *vector*; `across` is used as a preposition in this syntax. Iteration stops when there are no more elements in the specified array that can be referenced.

Some implementations might use a [user-supplied—GLS] `the` special operator in the *vector* form to produce more efficient code.

Example:

```
(loop for char across (the simple-string (find-message port))
      do (write-char char stream))
```

[Выражение цикла] **for** var [type-spec] `being` {`each` | `the`}
{`hash-key` | `hash-keys` | `hash-value` | `hash-values`}
{`in` | `of`} hash-table [`using` ({`hash-value` | `hash-key`} other-var)]
[Выражение цикла] **as** var [type-spec] `being` {`each` | `the`}
{`hash-key` | `hash-keys` | `hash-value` | `hash-values`}
{`in` | `of`} hash-table [`using` ({`hash-value` | `hash-key`} other-var)]

[This is the sixth of seven `for`/`as` syntaxes.—GLS]

This construct iterates over the elements, keys, and values of a hash table. The variable *var* takes on the value of each hash key or hash value in the specified hash table.

The following loop keywords serve as valid prepositions within this syntax.

`being` The keyword `being` marks the loop method to be used, either `hash-key` or `hash-value`.

`each`, `the` For purposes of readability, the loop keyword `each` should follow the loop keyword `being` when `hash-key` or `hash-value` is used. The loop keyword `the` is used with `hash-keys` and `hash-values`.

`hash-key`, `hash-keys` These loop keywords access each key entry of the hash table. If the name `hash-value` is specified in a `using` construct with one of these loop methods, the iteration can optionally access the keyed value. The order in which the keys are accessed is undefined; empty slots in the hash table are ignored.

`hash-value`, `hash-values` These loop keywords access each value entry of a hash table. If the name `hash-key` is specified in a `using` construct with one of these loop methods, the iteration can optionally access the key that corresponds to the value. The order in which the keys are accessed is undefined; empty slots in the hash table are ignored.

`using` The loop keyword `using` marks the optional key or the keyed value to be accessed. It allows you to access the hash key if iterating over the hash values, and the hash value if iterating over the hash keys.

`in, of` These loop prepositions mark the hash table *hash-table*.

Iteration stops when there are no more hash keys or hash values to be referenced in the specified hash table.

[Выражение цикла] **for** var [type-spec] `being` {`each` | `the`}
{`symbol` | `present-symbol` | `external-symbol` |
`symbols` | `present-symbols` | `external-symbols`}
{`in` | `of`} package
[Выражение цикла] **as** var [type-spec] `being` {`each` | `the`}
{`symbol` | `present-symbol` | `external-symbol` |
`symbols` | `present-symbols` | `external-symbols`}
{`in` | `of`} package
[This is the last of seven `for`/`as` syntaxes.—GLS]

This construct iterates over the symbols in a package. The variable *var* takes on the value of each symbol in the specified package.

The following loop keywords serve as valid prepositions within this syntax.

`being` The keyword `being` marks the loop method to be used: `symbol`, `present-symbol`, or `external-symbol`.

`each, the` For purposes of readability, the loop keyword `each` should follow the loop keyword `being` when `symbol`, `present-symbol`, or `external-symbol` is used. The loop keyword `the` is used with `symbols`, `present-symbols`, and `external-symbols`.

`present-symbol, present-symbols` These loop methods iterate over the symbols that are present but not external in a package. The package to be iterated over is specified in the same way that package arguments to the Common Lisp function `find-package` are specified. If you do not specify the package for the iteration, the current package is used. If you specify a package that does not exist, an error is signaled.

`symbol, symbols` These loop methods iterate over symbols that are accessible from a given package. The package to be iterated over is specified

in the same way that package arguments to the Common Lisp function `find-package` are specified. If you do not specify the package for the iteration, the current package is used. If you specify a package that does not exist, an error is signaled.

`external-symbol, external-symbols` These loop methods iterate over the external symbols of a package. The package to be iterated over is specified in the same way that package arguments to the Common Lisp function `find-package` are specified. If you do not specify the package for the iteration, the current package is used. If you specify a package that does not exist, an error is signaled.

`in, of` These loop prepositions mark the package *package*.

Iteration stops when there are no more symbols to be referenced in the specified package.

Example:

```
(loop for x being each present-symbol of "COMMON-LISP-USER"
    do (print x))                          ;Prints 7 lines in this example
COMMON-LISP-USER::IN
COMMON-LISP-USER::X
COMMON-LISP-USER::ALWAYS
COMMON-LISP-USER::FOO
COMMON-LISP-USER::Y
COMMON-LISP-USER::FOR
COMMON-LISP-USER::LUCID
  ⇒ NIL
```

[Выражение цикла] **repeat** expr

The `repeat` construct causes iteration to terminate after a specified number of times. The loop body is executed n times, where n is the value of the expression *expr*. The *expr* argument is evaluated one time in the loop prologue. If the expression evaluates to zero or to a negative number, the loop body is not evaluated.

The clause `repeat` n is roughly equivalent to a clause such as

for *internal-variable* downfrom (- n 1) to 0

but, in some implementations, the `repeat` construct might be more efficient.

Examples:

```
(loop repeat 3                                          ;Prints 3 lines
    do (format t "What I say three times is true~%"))
What I say three times is true
What I say three times is true
What I say three times is true
  ⇒ NIL

(loop repeat -15                                        ;Prints nothing
    do (format t "What you see is what you expect~%"))
  ⇒ NIL
```

25.7 End-Test Control

The loop keywords `always`, `never`, `thereis`, `until`, and `while` designate constructs that use a single test condition to determine when loop iteration should terminate.

The constructs `always`, `never`, and `thereis` provide specific values to be returned when a loop terminates. Using `always`, `never`, or `thereis` with value-returning accumulation clauses can produce unpredictable results. In all other respects these constructs behave like the `while` and `until` constructs.

The macro `loop-finish` can be used at any time to cause normal termination. In normal termination, `finally` clauses are executed and default return values are returned.

End-test control constructs can be used anywhere within the loop body. The termination conditions are tested in the order in which they appear.

[Выражение цикла] **while** expr
[Выражение цикла] **until** expr

The `while` construct allows iteration to continue until the specified expression *expr* evaluates to `nil`. The expression is re-evaluated at the location

of the `while` clause.

The `until` construct is equivalent to `while` (`not` *expr*). If the value of the specified expression is non-`nil`, iteration terminates.

You can use `while` and `until` at any point in a loop. If a `while` or `until` clause causes termination, any clauses that precede it in the source are still evaluated.

Examples:

```
;;; A classic "while-loop".
(loop while (hungry-p) do (eat))

;;; UNTIL NOT is equivalent to WHILE.
(loop until (not (hungry-p)) do (eat))

;;; Collect the length and the items of STACK.
(let ((stack '(a b c d e f)))
  (loop while stack
        for item = (length stack) then (pop stack)
        collect item))
  ⇒ (6 A B C D E F)

;;; Use WHILE to terminate a loop that otherwise wouldn't
;;; terminate. Note that WHILE occurs after the WHEN.
(loop for i fixnum from 3
      when (oddp i) collect i
      while (< i 5))
  ⇒ (3 5)
```

[Выражение цикла] **always** expr
[Выражение цикла] **never** expr
[Выражение цикла] **thereis** expr

The `always` construct takes one form and terminates the loop if the form ever evaluates to `nil`; in this case, it returns `nil`. Otherwise, it provides a default return value of `t`.

The `never` construct takes one form and terminates the loop if the form ever evaluates to non-`nil`; in this case, it returns `nil`. Otherwise, it provides a default return value of `t`.

The `thereis` construct takes one form and terminates the loop if the form ever evaluates to non-`nil`; in this case, it returns that value.

If the `while` or `until` construct causes termination, control is passed to the loop epilogue, where any `finally` clauses will be executed. Since `always`, `never`, and `thereis` use the Common Lisp macro `return` to terminate iteration, any `finally` clause that is specified is not evaluated.

Examples:

```
;;; Make sure I is always less than 11 (two ways).
;;; The FOR construct terminates these loops.
```

```
(loop for i from 0 to 10
      always (< i 11))
  ⇒ T
```

```
(loop for i from 0 to 10
      never (> i 11))
  ⇒ T
```

```
;;; If I exceeds 10, return I; otherwise, return NIL.
;;; The THEREIS construct terminates this loop.
```

```
(loop for i from 0
      thereis (when (> i 10) i) )
  ⇒ 11
```

```
;;; The FINALLY clause is not evaluated in these examples.

(loop for i from 0 to 10
      always (< i 9)
      finally (print "you won't see this"))
  ⇒ NIL

(loop never t
      finally (print "you won't see this"))
  ⇒ NIL

(loop thereis "Here is my value"
      finally (print "you won't see this"))
  ⇒ "Here is my value"

;;; The FOR construct terminates this loop,
;;; so the FINALLY clause is evaluated.

(loop for i from 1 to 10
      thereis (> i 11)
      finally (print i))                                ;Prints 1 line
11
  ⇒ NIL
```

```
(defstruct mountain height difficulty (why "because it is there"))
(setq everest (make-mountain :height '(2.86e-13 parsecs)))
(setq chocorua (make-mountain :height '(1059180001 microns)))
(defstruct desert area (humidity 0))
(setq sahara (make-desert :area '(212480000 square furlongs)))
      ;First there is a mountain, then there is no mountain, then there is ...
(loop for x in (list everest sahara chocorua)                    ; —GLS
    thereis (and (mountain-p x) (mountain-height x)))
  ⇒ (2.86E-13 PARSECS)

;;; If you could use this code to find a counterexample to
;;; Fermat's last theorem, it would still not return the value
;;; of the counterexample because all of the THEREIS clauses
;;; in this example return only T.  Of course, this code has
;;; never been observed to terminate.

(loop for z upfrom 2
    thereis
      (loop for n upfrom 3 below (log z 2)
          thereis
            (loop for x below z
                thereis
                  (loop for y below z
                      thereis (= (+ (expt x n)
                                    (expt y n))
                                 (expt z n))))))
```

[Макрос] **loop-finish**

The macro `loop-finish` terminates iteration normally and returns any accumulated result. If specified, a `finally` clause is evaluated.

In most cases it is not necessary to use `loop-finish` because other loop control clauses terminate the loop. Use `loop-finish` to provide a normal exit from a nested condition inside a loop.

You can use `loop-finish` inside nested Lisp code to provide a normal exit from a loop. Since `loop-finish` transfers control to the loop epilogue, using `loop-finish` within a `finally` expression can cause infinite looping.

Implementations are allowed to provide this construct as a local macro by using `macrolet`.

Examples:

```
;;; Print a date in February, but exclude leap day.
;;; LOOP-FINISH exits from the nested condition.
(loop for date in date-list
      do (case date
           (29 (when (eq month 'february)
                 (loop-finish))
            (format t "~:@(~A~) ~A" month date))))

;;; Terminate the loop, but return the accumulated count.
(loop for i in '(1 2 3 stop-here 4 5 6)
      when (symbolp i) do (loop-finish)
      count i)
  ⇒ 3

;;; This loop works just as well as the previous example.
(loop for i in '(1 2 3 stop-here 4 5 6)
      until (symbolp i)
      count i)
  ⇒ 3
```

25.8 Value Accumulation

Accumulating values during iteration and returning them from a loop is often useful. Some of these accumulations occur so frequently that special loop clauses have been developed to handle them.

The loop keywords `append`, `appending`, `collect`, `collecting`, `nconc`, and `nconcing` designate clauses that accumulate values in lists and return them.

The loop keywords `count`, `counting`, `maximize`, `maximizing`, `minimize`, `minimizing`, `sum`, and `summing` designate clauses that accumulate and return numerical values. [There is no semantic difference between the "ing" keywords and their non-"ing" counterparts. They are provided purely for the sake of stylistic diversity among users. I happen to prefer the non-"ing" forms—when I use `loop` at all.—GLS]

The loop preposition `into` can be used to name the variable used to hold partial accumulations. The variable is bound as if by the loop construct `with` (see section 25.9). If `into` is used, the construct does not provide a default return value; however, the variable is available for use in any `finally` clause.

You can combine value-returning accumulation clauses in a loop if all the clauses accumulate the same type of data object. By default, the Loop Facility returns only one value; thus, the data objects collected by multiple accumulation clauses as return values must have compatible types. For example, since both the `collect` and `append` constructs accumulate objects into a list that is returned from a loop, you can combine them safely.

```
;;; Collect every name and the kids in one list by using
;;; COLLECT and APPEND.
(loop for name in '(fred sue alice joe june)
      for kids in '((bob ken) () () (kris sunshine) ())
      collect name
      append kids)
  ⇒ (FRED BOB KEN SUE ALICE JOE KRIS SUNSHINE JUNE)
```

[In the preceding example, note that the items accumulated by the `collect` and `append` clauses are interleaved in the result list, according to the order in which the clauses were executed.—GLS]

Multiple clauses that do not accumulate the same type of data object can coexist in a loop only if each clause accumulates its values into a different user-specified variable. Any number of values can be returned from a loop if you use the Common Lisp function `values`, as the next example shows:

```
;;; Count and collect names and ages.
(loop for name in '(fred sue alice joe june)
      as age in '(22 26 19 20 10)
      append (list name age) into name-and-age-list
      count name into name-count
      sum age into total-age
      finally
        (return (values (round total-age name-count)
                        name-and-age-list)))
  ⇒ 19 and (FRED 22 SUE 26 ALICE 19 JOE 20 JUNE 10)
```

[Выражение цикла] **collect** expr [`into` var]

[Выражение цикла] **collecting** expr [`into` var]

During each iteration, these constructs collect the value of the specified expression into a list. When iteration terminates, the list is returned.

The argument *var* is set to the list of collected values; if *var* is specified, the loop does not return the final list automatically. If *var* is not specified, it is equivalent to specifying an internal name for *var* and returning its value in a `finally` clause. The *var* argument is bound as if by the construct `with`. You cannot specify a data type for *var*; it must be of type `list`.

Examples:

```
;;; Collect all the symbols in a list.
(loop for i in '(bird 3 4 turtle (1 . 4) horse cat)
      when (symbolp i) collect i)
  ⇒ (BIRD TURTLE HORSE CAT)

;;; Collect and return odd numbers.
(loop for i from 1 to 10
      if (oddp i) collect i)
  ⇒ (1 3 5 7 9)
```

```
;;; Collect items into local variable, but don't return them.
(loop for i in '(a b c d) by #'cddr
      collect i into my-list
      finally (print my-list))                          ;Prints 1 line
(A C)
  ⇒ NIL
```

[Выражение цикла] **append** expr [`into` var]
[Выражение цикла] **appending** expr [`into` var]
[Выражение цикла] **nconc** expr [`into` var]
[Выражение цикла] **nconcing** expr [`into` var]

These constructs are similar to `collect` except that the values of the specified expression must be lists.

The `append` keyword causes its list values to be concatenated into a single list, as if they were arguments to the Common Lisp function `append`.

The `nconc` keyword causes its list values to be concatenated into a single list, as if they were arguments to the Common Lisp function `nconc`. Note that the `nconc` keyword destructively modifies its argument lists.

The argument *var* is set to the list of concatenated values; if you specify *var*, the loop does not return the final list automatically. The *var* argument is bound as if by the construct `with`. You cannot specify a data type for *var*; it must be of type `list`.

Examples:

```
;;; Use APPEND to concatenate some sublists.
(loop for x in '((a) (b) ((c)))
      append x)
  ⇒ (A B (C))
```

```
;;; NCONC some sublists together. Note that only lists
;;; made by the call to LIST are modified.
(loop for i upfrom 0
      as x in '(a b (c))
      nconc (if (evenp i) (list x) '()))
  ⇒ (A (C))
```

[Выражение цикла] **count** expr [`into` var] [type-spec]
[Выражение цикла] **counting** expr [`into` var] [type-spec]

The `count` construct counts the number of times that the specified expression has a non-`nil` value.

The argument *var* accumulates the number of occurrences; if *var* is specified, the loop does not return the final count automatically. The *var* argument is bound as if by the construct `with`.

If `into` *var* is used, the optional *type-spec* argument specifies a data type for *var*. If there is no `into` variable, the optional *type-spec* argument applies to the internal variable that is keeping the count. In either case it is an error to specify a non-numeric data type. The default type is implementation-dependent, but it must be a subtype of `(or integer float)`.

Example:

```
(loop for i in '(a b nil c nil d e)
      count i)
  ⇒ 5
```

[Выражение цикла] **sum** expr [`into` var] [type-spec]
[Выражение цикла] **summing** expr [`into` var] [type-spec]

The `sum` construct forms a cumulative sum of the values of the specified expression at each iteration.

The argument *var* is used to accumulate the sum; if *var* is specified, the loop does not return the final sum automatically. The *var* argument is bound as if by the construct `with`.

If `into` *var* is used, the optional *type-spec* argument specifies a data type for *var*. If there is no `into` variable, the optional *type-spec* argument applies to the internal variable that is keeping the sum. In either case it is an error to specify a non-numeric data type. The default type is implementation-dependent, but it must be a subtype of `number`.

Examples:

```
;;; Sum the elements of a list.

(loop for i fixnum in '(1 2 3 4 5)
      sum i)
  ⇒ 15
```

```
;;; Sum a function of elements of a list.

(setq series
      '(1.2 4.3 5.7))
  ⇒ (1.2 4.3 5.7)

(loop for v in series
      sum (* 2.0 v))
  ⇒ 22.4
```

[Выражение цикла] **maximize** expr [`into` var] [type-spec]
[Выражение цикла] **maximizing** expr [`into` var] [type-spec]
[Выражение цикла] **minimize** expr [`into` var] [type-spec]
[Выражение цикла] **minimizing** expr [`into` var] [type-spec]

The `maximize` construct compares the value of the specified expression obtained during the first iteration with values obtained in successive iterations. The maximum value encountered is determined and returned. If the loop never executes the body, the returned value is not meaningful.

The `minimize` construct is similar to `maximize`; it determines and returns the minimum value.

The argument *var* accumulates the maximum or minimum value; if *var* is specified, the loop does not return the maximum or minimum automatically. The *var* argument is bound as if by the construct `with`.

If `into` *var* is used, the optional *type-spec* argument specifies a data type for *var*. If there is no `into` variable, the optional *type-spec* argument applies to the internal variable that is keeping the intermediate result. In either case it is an error to specify a non-numeric data type. The default type is implementation-dependent, but it must be a subtype of `(or integer float)`.

Examples:

```
(loop for i in '(2 1 5 3 4)
      maximize i)
  ⇒ 5
```

```
(loop for i in '(2 1 5 3 4)
      minimize i)
  ⇒ 1

;;; In this example, FIXNUM applies to the internal
;;; variable that holds the maximum value.

(setq series '(1.2 4.3 5.7))
  ⇒ (1.2 4.3 5.7)

(loop for v in series
      maximize (round v) fixnum)
  ⇒ 6

;;; In this example, FIXNUM applies to the variable RESULT.

(loop for v float in series
      minimize (round v) into result fixnum
      finally (return result))
  ⇒ 1
```

25.9 Variable Initializations

A local loop variable is one that exists only when the Loop Facility is invoked. At that time, the variables are declared and are initialized to some value. These local variables exist until loop iteration terminates, at which point they cease to exist. Implicitly variables are also established by iteration control clauses and the `into` preposition of accumulation clauses.

The loop keyword `with` designates a loop clause that allows you to declare and initialize variables that are local to a loop. The variables are initialized one time only; they can be initialized sequentially or in parallel.

By default, the `with` construct initializes variables sequentially; that is, one variable is assigned a value before the next expression is evaluated. How-

ever, by using the loop keyword `and` to join several `with` clauses, you can force initializations to occur in parallel; that is, all of the specified expressions are evaluated, and the results are bound to the respective variables simultaneously.

Use sequential binding for making the initialization of some variables depend on the values of previously bound variables. For example, suppose you want to bind the variables `a`, `b`, and `c` in sequence:

```
(loop with a = 1
      with b = (+ a 2)
      with c = (+ b 3)
      with d = (+ c 4)
      return (list a b c d))
  ⇒ (1 3 6 10)
```

The execution of the preceding loop is equivalent to the execution of the following code:

```
(let* ((a 1)
       (b (+ a 2))
       (c (+ b 3))
       (d (+ c 4)))
  (block nil
    (tagbody
      next-loop (return (list a b c d))
                (go next-loop)
      end-loop)))
```

If you are not depending on the value of previously bound variables for the initialization of other local variables, you can use parallel bindings as follows:

```
(loop with a = 1
       and b = 2
       and c = 3
       and d = 4
      return (list a b c d))
  ⇒ (1 2 3 4)
```

The execution of the preceding loop is equivalent to the execution of the following code:

```
(let ((a 1)
      (b 2)
      (c 3)
      (d 4))
  (block nil
    (tagbody
      next-loop (return (list a b c))
                (go next-loop)
      end-loop)))
```

[Выражение цикла] **with** var [type-spec] [= expr] {`and` var [type-spec] [= expr]}*

The `with` construct initializes variables that are local to a loop. The variables are initialized one time only.

If the optional *type-spec* argument is specified for any variable *var*, but there is no related expression *expr* to be evaluated, *var* is initialized to an appropriate default value for its data type. For example, for the data types `t`, `number`, and `float`, the default values are `nil`, `0`, and `0.0`, respectively. It is an error to specify a *type-spec* argument for *var* if the related expression returns a value that is not of the specified type. The optional `and` clause forces parallel rather than sequential initializations.

Examples:

```
;;; These bindings occur in sequence.
(loop with a = 1
      with b = (+ a 2)
      with c = (+ b 3)
      with d = (+ c 4)
      return (list a b c d))
  ⇒ (1 3 6 10)
```

```
;;; These bindings occur in parallel.
(setq a 5 b 10 c 1729)
(loop with a = 1
      and b = (+ a 2)
      and c = (+ b 3)
      and d = (+ c 4)
     return (list a b c d))
  ⇒ (1 7 13 1733)

;;; This example shows a shorthand way to declare
;;; local variables that are of different types.
(loop with (a b c) (float integer float)
     return (format nil "~A ~A ~A" a b c))
  ⇒ "0.0 0 0.0"

;;; This example shows a shorthand way to declare
;;; local variables that are of the same type.
(loop with (a b c) float
     return (format nil "~A ~A ~A" a b c))
  ⇒ "0.0 0.0 0.0"
```

25.10 Conditional Execution

The loop keywords `if`, `when`, and `unless` designate constructs that are useful when you want some loop clauses to operate under a specified condition.

If the specified condition is true, the succeeding loop clause is executed. If the specified condition is not true, the succeeding clause is skipped, and program control moves to the clause that follows the loop keyword `else`. If the specified condition is not true and no `else` clause is specified, the entire conditional construct is skipped. Several clauses can be connected into one compound clause with the loop keyword `and`. The end of the conditional clause can be marked with the keyword `end`.

[Выражение цикла] **if** expr clause {`and` clause}*
[`else` clause {`and` clause}*] [`end`]

[Выражение цикла] **when** expr clause {`and` clause}*
[`else` clause {`and` clause}*] [`end`]

[Выражение цикла] **unless** expr clause {`and` clause}*
[`else` clause {`and` clause}*] [`end`]

The constructs `when` and `if` allow conditional execution of loop clauses. These constructs are synonyms and can be used interchangeably. [Compare this to the *macro* `when`, which does not allow an "else" part.—GLS]

If the value of the test expression *expr* is non-`nil`, the expression *clause1* is evaluated. If the test expression evaluates to `nil` and an `else` construct is specified, the statements that follow the `else` are evaluated; otherwise, control passes to the next clause.

The `unless` construct is equivalent to `when` (`not` *expr*) and `if` (`not` *expr*). If the value of the test expression *expr* is `nil`, the expression *clause1* is evaluated. If the test expression evaluates to non-`nil` and an `else` construct is specified, the statements that follow the `else` are evaluated; otherwise, control passes to the next clause. [Compare this to the *macro* `unless`, which does not allow an "else" part—or do I mean a "then" part?! Ugh. To prevent confusion, I strongly recommend as a matter of style that `else` not be used with `unless` loop clauses.—GLS]

The *clause* arguments must be either accumulation, unconditional, or conditional clauses (see section 25.3.2). Clauses that follow the test expression can be grouped by using the loop keyword `and` to produce a compound clause.

The loop keyword `it` can be used to refer to the result of the test expression in a clause. If multiple clauses are connected with `and`, the `it` construct must be used in the first clause in the block. Since `it` is a loop keyword, `it` may not be used as a local variable within a loop.

If `when` or `if` clauses are nested, each `else` is paired with the closest preceding `when` or `if` construct that has no associated `else`.

The optional loop keyword `end` marks the end of the clause. If this keyword is not specified, the next loop keyword marks the end. You can use `end` to distinguish the scoping of compound clauses.

```
;;; Group conditional clauses into a block.
(loop for i in numbers-list
      when (oddp i)
        do (print i)
        and collect i into odd-numbers
        and do (terpri)
      else     ;I is even
        collect i into even-numbers
      finally
        (return (values odd-numbers even-numbers)))
```

```
;;; Collect numbers larger than 3.
(loop for i in '(1 2 3 4 5 6)
      when (and (> i 3) i)
      collect it)     ;it refers to (and (> i 3) i)
  ⇒ (4 5 6)
```

```
;;; Find a number in a list.
(loop for i in '(1 2 3 4 5 6)
      when (and (> i 3) i)
      return it)
  ⇒ 4
```

```
;;; The preceding example is similar to the following one.
(loop for i in '(1 2 3 4 5 6)
      thereis (and (> i 3) i))
  ⇒ 4
```

```
;;; An example of using UNLESS with ELSE (yuk).                    —GLS
(loop for turtle in teenage-mutant-ninja-turtles do
  (loop for x in '(joker brainiac shredder krazy-kat)
        unless (evil x)
          do (eat (make-pizza :anchovies t))
        else unless (and (eq x 'shredder) (attacking-p x))
             do (cut turtle slack) ;When the evil Shredder attacks,
           else (fight turtle x))) ; those turtle boys don't cut no slack



;;; Nest conditional clauses.
(loop for i in list
      when (numberp i)
        when (bignump i)
          collect i into big-numbers
        else     ;Not (bignump i)
          collect i into other-numbers
      else     ;Not (numberp i)
        when (symbolp i)
          collect i into symbol-list
        else     ;Not (symbolp i)
          (error "found a funny value in list ~S, value ~S~%"
                 "list i))

;;; Without the END marker, the last AND would apply to the
;;; inner IF rather than the outer one.
(loop for x from 0 to 3
      do (print x)
      if (zerop (mod x 2))
        do (princ " a")
          and if (zerop (floor x 2))
                 do (princ " b")
             end
          and do (princ " c"))
```

25.11 Unconditional Execution

The loop construct `do` (or `doing`) takes one or more expressions and simply evaluates them in order.

The loop construct `return` takes one expression and returns its value. It is equivalent to the clause `do (return` *value*`)`.

[Выражение цикла] **do** {expr}*
[Выражение цикла] **doing** {expr}*

The `do` construct simply evaluates the specified expressions wherever they occur in the expanded form of `loop`.

The *expr* argument can be any non-atomic Common Lisp form. Each *expr* is evaluated in every iteration.

The constructs `do`, `initially`, and `finally` are the only loop keywords that take an arbitrary number of forms and group them as if using an implicit `progn`. Because every loop clause must begin with a loop keyword, you would use the keyword `do` when no control action other than execution is required.

Examples:

```
;;; Print some numbers.
(loop for i from 1 to 5
      do (print i))                                  ;Prints 5 lines
1
2
3
4
5
  ⇒ NIL

;;; Print numbers and their squares.
;;; The DO construct applies to multiple forms.
(loop for i from 1 to 4
      do (print i)
         (print (* i i)))                            ;Prints 8 lines
```

```
1
1
2
4
3
9
4
16
  ⇒ NIL
```

[Выражение цикла] **return** expr

The `return` construct terminates a loop and returns the value of the specified expression as the value of the loop. This construct is similar to the Common Lisp special operator `return-from` and the Common Lisp macro `return`.

The Loop Facility supports the `return` construct for backward compatibility with older `loop` implementations. The `return` construct returns immediately and does not execute any `finally` clause that is given.

Examples:

```
;;; Signal an exceptional condition.
(loop for item in '(1 2 3 a 4 5)
      when (not (numberp item))
      return (cerror "enter new value"
                     "non-numeric value: ~s"
                     item))                                ;Signals an error
»Error: non-numeric value: A

;;; The previous example is equivalent to the following one.
(loop for item in '(1 2 3 a 4 5)
      when (not (numberp item))
      do (return
           (cerror "enter new value"
                   "non-numeric value: ~s"
                   item)))                                 ;Signals an error
»Error: non-numeric value: A
```

25.12 Miscellaneous Features

The Loop Facility provides the `named` construct to name a loop so that the Common Lisp special operator `return-from` can be used.

The loop keywords `initially` and `finally` designate loop constructs that cause expressions to be evaluated before and after the loop body, respectively.

The code for any `initially` clauses is collected into one `progn` in the order in which the clauses appeared in the loop. The collected code is executed once in the loop prologue after any implicit variable initializations.

The code for any `finally` clauses is collected into one `progn` in the order in which the clauses appeared in the loop. The collected code is executed once in the loop epilogue before any implicit values are returned from the accumulation clauses. Explicit returns in the loop body, however, will exit the loop without executing the epilogue code.

25.12.1 Data Types

Many loop constructs take a *type-spec* argument that allows you to specify certain data types for loop variables. While it is not necessary to specify a data type for any variable, by doing so you ensure that the variable has a correctly typed initial value. The type declaration is made available to the compiler for more efficient `loop` expansion. In some implementations, fixnum and float declarations are especially useful; the compiler notices them and emits more efficient code.

The *type-spec* argument has the following syntax:

type-spec ::= `of-type` *d-type-spec*
d-type-spec ::= *type-specifier* | `(`*d-type-spec* `.` *d-type-spec*`)`

A *type-specifier* in this syntax can be any Common Lisp type specifier. The *d-type-spec* argument is used for destructuring, as described in section 25.12.2. If the *d-type-spec* argument consists solely of the types `fixnum`, `float`, `t`, or `nil`, the `of-type` keyword is optional. The `of-type` construct is optional

in these cases to provide backward compatibility; thus the following two expressions are the same:

```
;;; This expression uses the old syntax for type specifiers.
(loop for i fixnum upfrom 3 ...)

;;; This expression uses the new syntax for type specifiers.
(loop for i of-type fixnum upfrom 3 ...)
```

25.12.2 Destructuring

Destructuring allows you to bind a set of variables to a corresponding set of values anywhere that you can normally bind a value to a single variable. During `loop` expansion, each variable in the variable list is matched with the values in the values list. If there are more variables in the variable list than there are values in the values list, the remaining variables are given a value of `nil`. If there are more values than variables listed, the extra values are discarded.

Suppose you want to assign values from a list to the variables `a`, `b`, and `c`. You could use one `for` clause to bind the variable `numlist` to the *car* of the specified expression, and then you could use another `for` clause to bind the variables `a`, `b`, and `c` sequentially.

```
;;; Collect values by using FOR constructs.
(loop for numlist in '((1 2 4.0) (5 6 8.3) (8 9 10.4))
      for a integer = (first numlist)
      and for b integer = (second numlist)
      and for c float = (third numlist)
      collect (list c b a))
  ⇒ ((4.0 2 1) (8.3 6 5) (10.4 9 8))
```

Destructuring makes this process easier by allowing the variables to be bound in parallel in each loop iteration. You can declare data types by using a list of *type-spec* arguments. If all the types are the same, you can use a shorthand destructuring syntax, as the second example following illustrates.

```
;;; Destructuring simplifies the process.
(loop for (a b c) (integer integer float) in
      '((1 2 4.0) (5 6 8.3) (8 9 10.4))
      collect (list c b a)))
  ⇒ ((4.0 2 1) (8.3 6 5) (10.4 9 8))

;;; If all the types are the same, this way is even simpler.
(loop for (a b c) float in
      '((1.0 2.0 4.0) (5.0 6.0 8.3) (8.0 9.0 10.4))
      collect (list c b a))
  ⇒ ((4.0 2.0 1.0) (8.3 6.0 5.0) (10.4 9.0 8.0))
```

If you use destructuring to declare or initialize a number of groups of variables into types, you can use the loop keyword `and` to simplify the process further.

```
;;; Initialize and declare variables in parallel
;;; by using the AND construct.
(loop with (a b) float = '(1.0 2.0)
      and (c d) integer = '(3 4)
      and (e f)
      return (list a b c d e f))
  ⇒ (1.0 2.0 3 4 NIL NIL)
```

A data type specifier for a destructuring pattern is a tree of type specifiers with the same shape as the tree of variables, with the following exceptions:

- When aligning the trees, an atom in the type specifier tree that matches a cons in the variable tree declares the same type for each variable.
- A cons in the type specifier tree that matches an atom in the variable tree is a non-atomic type specifer.

```
;;; Declare X and Y to be of type VECTOR and FIXNUM, respectively.
(loop for (x y) of-type (vector fixnum) in my-list do ...)
```

If `nil` is used in a destructuring list, no variable is provided for its place.

```
(loop for (a nil b) = '(1 2 3)
    do (return (list a b)))
  ⇒ (1 3)
```

Note that nonstandard lists can specify destructuring.

```
(loop for (x . y) = '(1 . 2)
    do (return y))
  ⇒ 2

(loop for ((a . b) (c . d))
        of-type ((float . float) (integer . integer))
        in '(((1.2 . 2.4) (3 . 4)) ((3.4 . 4.6) (5 . 6)))
    collect (list a b c d))
  ⇒ ((1.2 2.4 3 4) (3.4 4.6 5 6))
```

[It is worth noting that the destructuring facility of `loop` predates, and differs in some details from, that of `destructuring-bind`, an extension that has been provided by many implementors of Common Lisp.—GLS]

[Выражение цикла] **initially** {expr}*
[Выражение цикла] **finally** [`do` | `doing`] {expr}*
[Выражение цикла] **finally** `return` expr

The `initially` construct causes the specified expression to be evaluated in the loop prologue, which precedes all loop code except for initial settings specified by constructs `with`, `for`, or `as`. The `finally` construct causes the specified expression to be evaluated in the loop epilogue after normal iteration terminates.

The *expr* argument can be any non-atomic Common Lisp form.

Clauses such as `return`, `always`, `never`, and `thereis` can bypass the `finally` clause.

The Common Lisp macro `return` (or the `return` loop construct) can be used after `finally` to return values from a loop. The evaluation of the `return` form inside the `finally` clause takes precedence over returning the accumulation from clauses specified by such keywords as `collect`, `nconc`,

`append`, `sum`, `count`, `maximize`, and `minimize`; the accumulation values for these pre-empted clauses are not returned by the loop if `return` is used.

The constructs `do`, `initially`, and `finally` are the only loop keywords that take an arbitrary number of (non-atomic) forms and group them as if by using an implicit `progn`.

Examples:

```
;;; This example parses a simple printed string representation
;;; from BUFFER (which is itself a string) and returns the
;;; index of the closing double-quote character.

(loop initially (unless (char= (char buffer 0) #\")
                  (loop-finish))
      for i fixnum from 1 below (string-length buffer)
      when (char= (char buffer i) #\")
        return i)

;;; The FINALLY clause prints the last value of I.
;;; The collected value is returned.

(loop for i from 1 to 10
      when (> i 5)
        collect i
      finally (print i))                                ;Prints 1 line
11
  ⇒ (6 7 8 9 10)

;;; Return both the count of collected numbers
;;; as well as the numbers themselves.

(loop for i from 1 to 10
      when (> i 5)
        collect i into number-list
        and count i into number-count
      finally (return (values number-count number-list)))
  ⇒ 5 and (6 7 8 9 10)
```

[Выражение цикла] **named** name

The `named` construct allows you to assign a name to a `loop` construct so that you can use the Common Lisp special operator `return-from` to exit the named loop.

Only one name may be assigned per loop; the specified name becomes the name of the implicit block for the loop.

If used, the `named` construct must be the first clause in the loop expression, coming right after the word `loop`.

Example:

```
;;; Just name and return.
(loop named max
      for i from 1 to 10
      do (print i)
      do (return-from max 'done))                    ;Prints 1 line
1
  ⇒ DONE
```

Глава 26

Pretty Printing

Author: Richard C. Waters
preface: X3J13 voted in January 1989 to adopt a facility for user-controlled pretty printing as a part of the forthcoming draft Common Lisp standard. This facility is the culmination of thirteen years of design, testing, revision, and use of this approach.

This chapter presents the bulk of the Common Lisp pretty printing specification, written by Richard C. Waters. I have edited it only very lightly to conform to the overall style of this book.

—Guy L. Steele Jr.

26.1 Introduction

Pretty printing has traditionally been a black box process, displaying program code using a set of fixed layout rules. Its utility can be greatly enhanced by opening it up to user control. The facilities described in this chapter provide general and powerful means for specifying pretty-printing behavior.

By providing direct access to the mechanisms within the pretty printer that make dynamic decisions about layout, the macros and functions `pprint-logical-block`, `pprint-newline`, and `pprint-indent` make it possible to specify pretty printing layout rules as a part of any function that produces output. They also make it very easy for the function to support detection of circularity and sharing and abbreviation based on length and nesting depth. Using the function `set-pprint-dispatch`, one can associate a user-defined pretty printing function with any type of object. A small

set of new `format` directives allows concise implementation of user-defined pretty-printing functions. Together, these facilities enable users to redefine the way code is displayed and allow the full power of pretty printing to be applied to complex combinations of data structures.

Заметка для реализации: This chapter describes the interface of the XP pretty printer. XP is described fully in [54], which also explains how to obtain a portable implementation. XP uses a highly efficient linear-time algorithm. When properly integrated into a Common Lisp, this algorithm supports pretty printing that is only fractionally slower than ordinary printing.

26.2 Pretty Printing Control Variables

The function `write` accepts keyword arguments named `:pprint-dispatch`, `:miser-width`, `:right-margin`, and `:lines`, corresponding to these variables.

[Variable] ***print-pprint-dispatch***

When `*print-pretty*` is not `nil`, printing is controlled by the 'pprint dispatch table' stored in the variable `*print-pprint-dispatch*`. The initial value of `*print-pprint-dispatch*` is implementation-dependent and causes traditional pretty printing of Lisp code. The last section of this chapter explains how the contents of this table can be changed.

[Variable] ***print-right-margin***

A primary goal of pretty printing is to keep the output between a pair of margins. The left margin is set at the column where the output begins. If this cannot be determined, the left margin is set to zero.

When `*print-right-margin*` is not `nil`, it specifies the right margin to use when making layout decisions. When `*print-right-margin*` is `nil` (the initial value), the right margin is set at the maximum line length that can be displayed by the output stream without wraparound or truncation. If this cannot be determined, the right margin is set to an implementation-dependent value.

To allow for the possibility of variable-width fonts, `*print-right-margin*` is in units of ems—the width of an "m" in the font being used to display characters on the relevant output stream at the moment when the variables are consulted.

[Variable] ***print-miser-width***

If `*print-miser-width*` is not `nil`, the pretty printer switches to a compact style of output (called miser style) whenever the width available for printing a substructure is less than or equal to `*print-miser-width*` ems. The initial value of `*print-miser-width*` is implementation-dependent.

[Variable] ***print-lines***

When given a value other than its initial value of `nil`, `*print-lines*` limits the number of output lines produced when something is pretty printed. If an attempt is made to go beyond `*print-lines*` lines, " .." (a space and two periods) is printed at the end of the last line followed by all of the suffixes (closing delimiters) that are pending to be printed.

```
(let ((*print-right-margin* 25) (*print-lines* 3))
  (pprint '(progn (setq a 1 b 2 c 3 d 4))))

(PROGN (SETQ A 1
             B 2
             C 3 ..))
```

(The symbol ".." is printed out to ensure that a reader error will occur if the output is later read. A symbol different from "..." is used to indicate that a different kind of abbreviation has occurred.)

26.3 Dynamic Control of the Arrangement of Output

The following functions and macros support precise control of what should be done when a piece of output is too large to fit in the space available. Three concepts underlie the way these operations work: *logical blocks*, *conditional newlines*, and *sections*. Before proceeding further, it is important to define these terms.

The first line of figure 26.1 shows a schematic piece of output. The characters in the output are represented by hyphens. The positions of conditional

newlines are indicated by digits. The beginnings and ends of logical blocks are indicated in the figure by "<" and ">" respectively.

The output as a whole is a logical block and the outermost section. This section is indicated by the 0's on the second line of figure 26.1. Logical blocks nested within the output are specified by the macro `pprint-logical-block`. Conditional newline positions are specified by calls on `pprint-newline`. Each conditional newline defines two sections (one before it and one after it) and is associated with a third (the section immediately containing it).

The section after a conditional newline consists of all the output up to, but not including, (a) the next conditional newline immediately contained in the same logical block; or if (a) is not applicable, (b) the next newline that is at a lesser level of nesting in logical blocks; or if (b) is not applicable, (c) the end of the output.

The section before a conditional newline consists of all the output back to, but not including, (a) the previous conditional newline that is immediately contained in the same logical block; or if (a) is not applicable, (b) the beginning of the immediately containing logical block. The last four lines in figure 26.1 indicate the sections before and after the four conditional newlines.

The section immediately containing a conditional newline is the shortest section that contains the conditional newline in question. In figure 26.1, the first conditional newline is immediately contained in the section marked with 0's, the second and third conditional newlines are immediately contained in the section before the fourth conditional newline, and the fourth conditional newline is immediately contained in the section after the first conditional newline.

Whenever possible, the pretty printer displays the entire contents of a section on a single line. However, if the section is too long to fit in the space available, line breaks are inserted at conditional newline positions within the section.

[Function] **pprint-newline** *kind* **&optional** *stream*

The *stream* (which defaults to `*standard-output*`) follows the standard conventions for stream arguments to printing functions (that is, `nil` stands for `*standard-output*` and `t` stands for `*terminal-io*`). The *kind* argument specifies the style of conditional newline. It must be one of `:linear`, `:fill`, `:miser`, or `:mandatory`. An error is signaled if any

Изображение 26.1: Example of Logical Blocks, Conditional Newlines, and Sections

```
<-1—<–<–2—3->–4–>->
00000000000000000000000000
11 1111111111111111111111
        22 222
           333 3333
      44444444444444 44444
```

other value is supplied. If *stream* is a pretty printing stream created by `pprint-logical-block`, a line break is inserted in the output when the appropriate condition below is satisfied. Otherwise, `pprint-newline` has no effect. The value `nil` is always returned.

If *kind* is `:linear`, it specifies a 'linear-style' conditional newline. A line break is inserted if and only if the immediately containing section cannot be printed on one line. The effect of this is that line breaks are either inserted at every linear-style conditional newline in a logical block or at none of them.

If *kind* is `:miser`, it specifies a 'miser-style' conditional newline. A line break is inserted if and only if the immediately containing section cannot be printed on one line and miser style is in effect in the immediately containing logical block. The effect of this is that miser-style conditional newlines act like linear-style conditional newlines, but only when miser style is in effect. Miser style is in effect for a logical block if and only if the starting position of the logical block is less than or equal to `*print-miser-width*` from the right margin.

If *kind* is `:fill`, it specifies a 'fill-style' conditional newline. A line break is inserted if and only if either (a) the following section cannot be printed on the end of the current line, (b) the preceding section was not printed on a single line, or (c) the immediately containing section cannot be printed on one line and miser style is in effect in the immediately containing logical block. If a logical block is broken up into a number of subsections by fill-style conditional newlines, the basic effect is that the logical block is printed with as many subsections as possible on each line. However, if miser style is in effect, fill-style conditional newlines act like linear-style conditional newlines.

If *kind* is `:mandatory`, it specifies a 'mandatory-style' conditional newline. A line break is always inserted. This implies that none of the containing sections can be printed on a single line and will therefore trigger the insertion of line breaks at linear-style conditional newlines in these sections.

When a line break is inserted by any type of conditional newline, any blanks that immediately precede the conditional newline are omitted from the output and indentation is introduced at the beginning of the next line. By default, the indentation causes the following line to begin in the same horizontal position as the first character in the immediately containing logical block. (The indentation can be changed via `pprint-indent`.)

There are a variety of ways *un*conditional newlines can be introduced into the output (for example, via `terpri` or by printing a string containing a newline character). As with mandatory conditional newlines, this prevents any of the containing sections from being printed on one line. In general, when an unconditional newline is encountered, it is printed out without suppression of the preceding blanks and without any indentation following it. However, if a per-line prefix has been specified (see `pprint-logical-block`), that prefix will always be printed no matter how a newline originates.

[Макрос] **pprint-logical-block** (stream-symbol list
[[{`:prefix` | `:per-line-prefix`} p | `:suffix` s]])
{form}*

This macro causes printing to be grouped into a logical block. It returns `nil`.

The *stream-symbol* must be a symbol. If it is `nil`, it is treated the same as if it were `*standard-output*`. If it is `t`, it is treated the same as if it were `*terminal-io*`. The run-time value of *stream-symbol* must be a stream (or `nil` standing for `*standard-output*` or `t` standing for `*terminal-io*`). The logical block is printed into this destination stream.

The body (which consists of the *form*s) can contain any arbitrary Lisp forms. Within the body, *stream-symbol* (or `*standard-output*` if *stream-symbol* is `nil`, or `*terminal-io*` if *stream-symbol* is `t`) is bound to a "pretty printing" stream that supports decisions about the arrangement of output and then forwards the output to the destination stream. All the standard printing functions (for example, `write`, `princ`, `terpri`) can be used to send output to the pretty printing stream created by `pprint-logical-block`. All and only the output sent to this pretty printing stream is treated as being in the logical block.

`pprint-logical-block` and the pretty printing stream it creates have dynamic extent. It is undefined what happens if output is attempted outside of this extent to the pretty printing stream created. It is unspecified what happens if, within this extent, any output is sent directly to the underlying destination stream (by calling `write-char`, for example).

The `:suffix`, `:prefix`, and `:per-line-prefix` arguments must all be expressions that (at run time) evaluate to strings. The `:suffix` argument *s* (which defaults to the null string) specifies a suffix that is printed just after the logical block. The `:prefix` and `:per-line-prefix` arguments are mutually exclusive. If neither `:prefix` nor `:per-line-prefix` is specified, a `:prefix` of the null string is assumed. The `:prefix` argument specifies a prefix *p* that is printed before the beginning of the logical block. The `:per-line-prefix` specifies a prefix *p* that is printed before the block and at the beginning of each subsequent line in the block. An error is signaled if `:prefix` and `:per-line-prefix` are both used or if a `:suffix`, `:prefix`, or `:pre-line-prefix` argument does not evaluate to a string.

The *list* is interpreted as being a list that the body is responsible for printing. (See `pprint-exit-if-list-exhausted` and `pprint-pop`.) If *list* does not (at run time) evaluate to a list, it is printed using `write`. (This makes it easier to write printing functions that are robust in the face of malformed arguments.) If `*print-circle*` (and possibly also `*print-shared*`) is not `nil` and *list* is a circular (or shared) reference to a cons, then an appropriate "`#`*n*`#`" marker is printed. (This makes it easy to write printing functions that provide full support for circularity and sharing abbreviation.) If `*print-level*` is not `nil` and the logical block is at a dynamic nesting depth of greater than `*print-level*` in logical blocks, "`#`" is printed. (This makes it easy to write printing functions that provide full support for depth abbreviation.)

If any of the three preceding conditions occurs, the indicated output is printed on *stream-symbol* and the *body* is skipped along with the printing of the prefix and suffix. (If the body is not responsible for printing a list, then the first two tests above can be turned off by supplying `nil` for the *list* argument.)

In addition to the *list* argument of `pprint-logical-block`, the arguments of the standard printing functions such as `write`, `print`, `pprint`, `print1`, and `pprint`, as well as the arguments of the standard `format` directives such as `~A`, `~S`, (and `~W`) are all checked (when necessary) for circularity and sharing. However, such checking is not applied to the arguments of the

functions `write-line`, `write-string`, and `write-char` or to the literal text output by `format`. A consequence of this is that you must use one of the latter functions if you want to print some literal text in the output that is not supposed to be checked for circularity or sharing. (See the examples below.)

Заметка для реализации: Detection of circularity and sharing is supported by the pretty printer by in essence performing the requested output twice. On the first pass, circularities and sharing are detected and the actual outputting of characters is suppressed. On the second pass, the appropriate "`#`*n*`=`" and "`#`*n*`#`" markers are inserted and characters are output.

A consequence of this two-pass approach to the detection of circularity and sharing is that the body of a `pprint-logical-block` must not perform any side-effects on the surrounding environment. This includes not modifying any variables that are bound outside of its scope. Obeying this restriction is facilitated by using `pprint-pop`, instead of an ordinary `pop` when traversing a list being printed by the body of a `pprint-logical-block`.)

[Макрос] **pprint-exit-if-list-exhausted**

`pprint-exit-if-list-exhausted` tests whether or not the *list* argument of `pprint-logical-block` has been exhausted (see `pprint-pop`). If this list has been reduced to `nil`, `pprint-exit-if-list-exhausted` terminates the execution of the immediately containing `pprint-logical-block` except for the printing of the suffix. Otherwise `pprint-exit-if-list-exhausted` returns `nil`. An error message is issued if `pprint-exit-if-list-exhausted` is used anywhere other than syntactically nested within a call on `pprint-logical-block`. It is undefined what happens if `pprint-pop` is executed outside of the dynamic extent of this `pprint-logical-block`.

[Макрос] **pprint-pop**

`pprint-pop` pops elements one at a time off the *list* argument of `pprint-logical-block`, taking care to obey `*print-length*`, `*print-circle*`, and `*print-shared*`. An error message is issued if it is used anywhere other than syntactically nested within a call on `pprint-logical-block`. It is undefined what happens if `pprint-pop` is executed outside of the dynamic extent of this call on `pprint-logical-block`.

Each time `pprint-pop` is called, it pops the next value off the *list* argument of `pprint-logical-block` and returns it. However, before doing this, it performs three tests. If the remaining list is not a list (neither a cons nor `nil`), "`. `" is printed followed by the remaining list. (This

makes it easier to write printing functions that are robust in the face of malformed arguments.) If `*print-length*` is `nil` and `pprint-pop` has already been called `*print-length*` times within the immediately containing logical block, "`...`" is printed. (This makes it easy to write printing functions that properly handle `*print-length*`.) If `*print-circle*` (and possibly also `*print-shared*`) is not `nil`, and the remaining list is a circular (or shared) reference, then "`. `" is printed followed by an appropriate "`#n#`" marker. (This catches instances of cdr circularity and sharing in lists.)

If any of the three preceding conditions occurs, the indicated output is printed on the pretty printing stream created by the immediately containing `pprint-logical-block` and the execution of the immediately containing `pprint-logical-block` is terminated except for the printing of the suffix.

If `pprint-logical-block` is given a *list* argument of `nil`—because it is not processing a list—`pprint-pop` can still be used to obtain support for `*print-length*` (see the example function `pprint-vector` below). In this situation, the first and third tests above are disabled and `pprint-pop` always returns `nil`.

[Function] **pprint-indent** *relative-to n* **&optional** *stream*

`pprint-indent` specifies the indentation to use in a logical block. *Stream* (which defaults to `*standard-output*`) follows the standard conventions for stream arguments to printing functions. The argument *n* specifies the indentation in ems. If *relative-to* is `:block`, the indentation is set to the horizontal position of the first character in the block plus *n* ems. If *relative-to* is `:current`, the indentation is set to the current output position plus *n* ems.

The argument *n* can be negative; however, the total indentation cannot be moved left of the beginning of the line or left of the end of the rightmost per-line prefix. Changes in indentation caused by `pprint-indent` do not take effect until after the next line break. In addition, in miser mode all calls on `pprint-indent` are ignored, forcing the lines corresponding to the logical block to line up under the first character in the block.

An error is signaled if a value other than `:block` or `:current` is supplied for *relative-to*. If *stream* is a pretty printing stream created by `pprint-logical-block`, `pprint-indent` sets the indentation in the innermost dynamically enclosing logical block. Otherwise, `pprint-indent` has no effect. The value `nil` is always returned.

[Function] **pprint-tab** *kind colnum colinc* **&optional** *stream*

`pprint-tab` specifies tabbing as performed by the standard `format` directive `~T`. *Stream* (which defaults to `*standard-output*`) follows the standard conventions for stream arguments to printing functions. The arguments *colnum* and *colinc* correspond to the two parameters to `~T` and are in terms of ems. The *kind* argument specifies the style of tabbing. It must be one of `:line` (tab as by `~T`) `:section` (tab as by `~T`, but measuring horizontal positions relative to the start of the dynamically enclosing section), `:line-relative` (tab as by `~@T`), or `:section-relative` (tab as by `~@T`, but measuring horizontal positions relative to the start of the dynamically enclosing section). An error is signaled if any other value is supplied for *kind*. If *stream* is a pretty printing stream created by `pprint-logical-block`, tabbing is performed. Otherwise, `pprint-tab` has no effect. The value `nil` is always returned.

[Function] **pprint-fill** *stream list* **&optional** *colon? atsign?*
[Function] **pprint-linear** *stream list* **&optional** *colon? atsign?*
[Function] **pprint-tabular** *stream list* **&optional** *colon? atsign? tabsize*

These three functions specify particular ways of pretty printing lists. *Stream* follows the standard conventions for stream arguments to printing functions. Each function prints parentheses around the output if and only if *colon?* (default `t`) is not `nil`. Each function ignores its *atsign?* argument and returns `nil`. (These two arguments are included in this way so that these functions can be used via `~/.../` and as `set-pprint-dispatch` functions as well as directly.) Each function handles abbreviation and the detection of circularity and sharing correctly and uses `write` to print *list* when given a non-list argument.

The function `pprint-linear` prints a list either all on one line or with each element on a separate line. The function `pprint-fill` prints a list with as many elements as possible on each line. The function `pprint-tabular` is the same as `pprint-fill` except that it prints the elements so that they line up in columns. This function takes an additional argument `tabsize` (default 16) that specifies the column spacing in ems.

As an example of the interaction of logical blocks, conditional newlines, and indentation, consider the function `pprint-defun` below. This function pretty prints a list whose *car* is `defun` in the standard way assuming that the length of the list is exactly 4.

```
;;; Pretty printer function for DEFUN forms.

(defun pprint-defun (list)
  (pprint-logical-block (nil list :prefix "(" :suffix ")")
    (write (first list))
    (write-char #\space)
    (pprint-newline :miser)
    (pprint-indent :current 0)
    (write (second list))
    (write-char #\space)
    (pprint-newline :fill)
    (write (third list))
    (pprint-indent :block 1)
    (write-char #\space)
    (pprint-newline :linear)
    (write (fourth list))))
```

Suppose that one evaluates the following:

```
(pprint-defun '(defun prod (x y) (* x y)))
```

If the line width available is greater than or equal to 26, all of the output appears on one line. If the width is reduced to 25, a line break is inserted at the linear-style conditional newline before `(* X Y)`, producing the output shown below. The `(pprint-indent :block 1)` causes `(* X Y)` to be printed at a relative indentation of 1 in the logical block.

```
(DEFUN PROD (X Y)
  (* X Y))
```

If the width is 15, a line break is also inserted at the fill-style conditional newline before the argument list. The argument list lines up under the function name because of the call on `(pprint-indent :current 0)` before the printing of the function name.

```
(DEFUN PROD
       (X Y)
  (* X Y))
```

If `*print-miser-width*` were greater than or equal to 14, the output would have been entirely in miser mode. All indentation changes are ignored in miser mode and line breaks are inserted at miser-style conditional newlines. The result would have been as follows:

```
(DEFUN
 PROD
 (X Y)
 (* X Y))
```

As an example of the use of a per-line prefix, consider that evaluating the expression

```
(pprint-logical-block (nil nil :per-line-prefix ";;; ")
  (pprint-defun '(defun prod (x y) (* x y))))
```

produces the output

```
;;; (DEFUN PROD
;;;        (X Y)
;;;   (* X Y))
```

with a line width of 20 and `nil` as the value of the printer control variable `*print-miser-width*`.

(If `*print-miser-width*` were not `nil` the output

```
;;; (DEFUN
;;;  PROD
;;;  (X Y)
;;;  (* X Y))
```

might appear instead.)

As a more complex (and realistic) example, consider the function `pprint-let` below. This specifies how to pretty print a `let` in the standard style. It is more complex than `pprint-defun` because it has to deal with nested structure. Also, unlike `pprint-defun`, it contains complete code to print readably any possible list that begins with the symbol `let`. The outermost `pprint-logical-block` handles the printing of the input list as a whole and specifies that parentheses should be printed in the output. The second `pprint-logical-block` handles the list of binding pairs. Each pair in the list is itself printed by the innermost `pprint-logical-block`. (A `loop` is used instead of merely decomposing the pair into two elements so that readable output will be produced no matter whether the list corresponding to the pair has one element, two elements, or (being malformed) has more than two elements.) A space and a fill-style conditional newline are placed after each pair except the last. The loop at the end of the topmost `pprint-logical-block` prints out the forms in the body of the `let` separated by spaces and linear-style conditional newlines.

```
;;; Pretty printer function for LET forms,
;;; carefully coded to handle malformed binding pairs.

(defun pprint-let (list)
  (pprint-logical-block (nil list :prefix "(" :suffix ")")
    (write (pprint-pop))
    (pprint-exit-if-list-exhausted)
    (write-char #\space)
    (pprint-logical-block
       (nil (pprint-pop) :prefix "(" :suffix ")")
      (pprint-exit-if-list-exhausted)
      (loop (pprint-logical-block
              (nil (pprint-pop) :prefix "(" :suffix ")")
            (pprint-exit-if-list-exhausted)
            (loop (write (pprint-pop))
                  (pprint-exit-if-list-exhausted)
                  (write-char #\space)
                  (pprint-newline :linear)))
          (pprint-exit-if-list-exhausted)
          (write-char #\space)
          (pprint-newline :fill)))
    (pprint-indent :block 1)
    (loop (pprint-exit-if-list-exhausted)
          (write-char #\space)
          (pprint-newline :linear)
          (write (pprint-pop)))))
```

Suppose that the following is evaluated with `*print-level*` having the value `4` and `*print-circle*` having the value `t`.

```
(pprint-let '#1=(let (x (*print-length* (f (g 3)))
                      (z . 2) (k (car y)))
                  (setq x (sqrt z)) #1#))
```

If the line length is greater than or equal to 77, the output produced appears on one line. However, if the line length is 76, line breaks are inserted at the linear-style conditional newlines separating the forms in the body and

the output below is produced. Note that the degenerate binding pair `X` is printed readably even though it fails to be a list; a depth abbreviation marker is printed in place of `(G 3)`; the binding pair `(Z . 2)` is printed readably even though it is not a proper list; and appropriate circularity markers are printed.

```
#1=(LET (X (*PRINT-LENGTH* (F #)) (Z . 2) (K (CAR Y)))
    (SETQ X (SQRT Z))
    #1#)
```

If the line length is reduced to 35, a line break is inserted at one of the fill-style conditional newlines separating the binding pairs.

```
#1=(LET (X (*PRINT-PRETTY* (F #))
       (Z . 2) (K (CAR Y)))
    (SETQ X (SQRT Z))
    #1#)
```

Suppose that the line length is further reduced to 22 and `*print-length*` is set to 3. In this situation, line breaks are inserted after both the first and second binding pairs. In addition, the second binding pair is itself broken across two lines. Clause (b) of the description of fill-style conditional newlines prevents the binding pair `(Z . 2)` from being printed at the end of the third line. Note that the length abbreviation hides the circularity from view and therefore the printing of circularity markers disappears.

```
(LET (X
     (*PRINT-LENGTH*
      (F #))
     (Z . 2) ...)
  (SETQ X (SQRT Z))
  ...)
```

The function `pprint-tabular` could be defined as follows:

```
(defun pprint-tabular (s list &optional (c? t) a? (size 16))
  (declare (ignore a?))
  (pprint-logical-block
      (s list :prefix (if c? "(" "") :suffix (if c? ")" ""))
    (pprint-exit-if-list-exhausted)
    (loop (write (pprint-pop) :stream s)
          (pprint-exit-if-list-exhausted)
          (write-char #\space s)
          (pprint-tab :section-relative 0 size s)
          (pprint-newline :fill s))))
```

Evaluating the following with a line length of 25 produces the output shown.

```
(princ "Roads ")
(pprint-tabular nil '(elm main maple center) nil nil 8)

Roads ELM     MAIN
      MAPLE   CENTER
```

The function below prints a vector using `#(...)` notation.

```
(defun pprint-vector (v)
  (pprint-logical-block (nil nil :prefix "#(" :suffix ")")
    (let ((end (length v)) (i 0))
      (when (plusp end)
        (loop (pprint-pop)
              (write (aref v i))
              (if (= (incf i) end) (return nil))
              (write-char #\space)
              (pprint-newline :fill))))))
```

Evaluating the following with a line length of 15 produces the output shown.

```
(pprint-vector '#(12 34 567 8 9012 34 567 89 0 1 23))

#(12 34 567 8
  9012 34 567
  89 0 1 23)
```

26.4 Format Directive Interface

The primary interface to operations for dynamically determining the arrangement of output is provided through the functions above. However, an additional interface is provided via a set of format directives because, as shown by the examples in this section and the next, `format` strings are typically a much more compact way to specify pretty printing. In addition, without such an interface, one would have to abandon the use of `format` when interacting with the pretty printer.

`~W` *Write.* An *arg*, any Lisp object, is printed obeying *every* printer control variable (as by `write`). In addition, `~W` interacts correctly with depth abbreviation by not resetting the depth counter to zero. `~W` does not accept parameters. If given the colon modifier, `~W` binds `*print-pretty*` to `t`. If given the atsign modifier, `~W` binds `*print-level*` and `*print-length*` to `nil`.

`~W` provides automatic support for circularity detection. If `*print-circle*` (and possibly also `*print-shared*`) is not `nil` and `~W` is applied to an argument that is a circular (or shared) reference, an appropriate "`#`*n*`#`" marker is inserted in the output instead of printing the argument.

`~_` *Conditional newline.* Without any modifiers, `~_` is equivalent to `(pprint-newline :linear)`. The directive `~@_` is equivalent to `(pprint-newline :miser)`. The directive `~:_` is equivalent to

`(pprint-newline :fill)`. The directive `~:@_` is equivalent to `(pprint-newline :mandatory)`.

`~<`*str*`~:>` *Logical block.* If `~:>` is used to terminate a `~<...` directive, the directive is equivalent to a call on `pprint-logical-block`. The `format` argument corresponding to the `~<...~:>` directive is treated in the same way as the *list* argument to `pprint-logical-block`, thereby providing automatic support for non-list arguments and the detection of circularity, sharing, and depth abbreviation. The portion of the `format` control string nested within the `~<...~:>` specifies the `:prefix` (or `:per-line-prefix`), `:suffix`, and body of the `pprint-logical-block`.

The `format` string portion enclosed by `~<...~:>` can be divided into segments `~<`*prefix*`~;`*body*`~;` *suffix*`~:>` by `~;` directives. If the first section is terminated by `~@;`, it specifies a per-line prefix rather than a simple prefix. The prefix and suffix cannot contain `format` directives. An error is signaled if either the prefix or suffix fails to be a constant string or if the enclosed portion is divided into more than three segments.

If the enclosed portion is divided into only two segments, the suffix defaults to the null string. If the enclosed portion consists of only a single segment, both the prefix and the suffix default to the null string. If the colon modifier is used (that is, `~:<...~:>`), the prefix and suffix default to `"("` and `")"`, respectively, instead of the null string.

The body segment can be any arbitrary `format` control string. This `format` control string is applied to the elements of the list corresponding to the `~<...~:>` directive as a whole. Elements are extracted from this list using `pprint-pop`, thereby providing automatic support for malformed lists and the detection of circularity, sharing, and length abbreviation. Within the body segment, `~^` acts like `pprint-exit-if-list-exhausted`.

`~<...~:>` supports a feature not supported by `pprint-logical-block`. If `~:@>` is used to terminate the directive (that is, `~<...~:@>`), then a fill-style conditional newline is automatically inserted after each group of blanks immediately contained in the body (except for blanks after a `~<newline>` directive). This makes it easy to achieve the equivalent of paragraph filling.

If the atsign modifier is used with ~<...~:>, the entire remaining argument list is passed to the directive as its argument. All of the remaining arguments are always consumed by ~@<...~:>, even if they are not all used by the format string nested in the directive. Other than the difference in its argument, ~@<...~:> is exactly the same as ~<...~:>, except that circularity (and sharing) detection is not applied if the ~@<...~:> is at top level in a format string. This ensures that circularity detection is applied only to data lists and not to format argument lists.

To a considerable extent, the basic form of the directive ~<...~> is incompatible with the dynamic control of the arrangement of output by ~W, ~_, ~<...~:>, ~I, and ~:T. As a result, an error is signaled if any of these directives is nested within ~<...~>. Beyond this, an error is also signaled if the ~<...~:;...~> form of ~<...~> is used in the same format string with ~W, ~_, ~<...~:>, ~I, or ~:T.

~I *Indent.* ~*n*I is equivalent to (pprint-indent :block *n*). ~:*n*I is equivalent to (pprint-indent :current *n*). In both cases, *n* defaults to zero if it is omitted.

~:T *Tabulate.* If the colon modifier is used with the ~T directive, the tabbing computation is done relative to the column where the section immediately containing the directive begins, rather than with respect to column zero. ~*n*,*m*:T is equivalent to (pprint-tab :section *n m*). ~*n*,*m*:@T is equivalent to (pprint-tab :section-relative *n m*). The numerical parameters are both interpreted as being in units of ems and both default to 1.

~/*name*/ *Call function.* User-defined functions can be called from within a format string by using the directive ~/*name*/. The colon modifier, the atsign modifier, and arbitrarily many parameters can be specified with the ~/*name*/ directive. The *name* can be any string that does not contain "/". All of the characters in *name* are treated as if they were upper case. If *name* contains a ":" or "::", then everything up to but not including the first ":" or "::" is taken to be a string that names a package. Everything after the first ":" or "::" (if any) is taken to be a string that names a symbol. The function corresponding to a ~/*name*/ directive is obtained by looking up the symbol that has the indicated

name in the indicated package. If *name* does not contain a ":" or "::", then the whole name string is looked up in the `user` package.

When a `~/`*name*`/` directive is encountered, the indicated function is called with four or more arguments. The first four arguments are the output stream, the `format` argument corresponding to the directive, the value `t` if the colon modifier was used (`nil` otherwise), and the value `t` if the atsign modifier was used (`nil` otherwise). The remaining arguments consist of any parameters specified with the directive. The function should print the argument appropriately. Any values returned by the function are ignored.

The three functions `pprint-linear`, `pprint-fill`, and `pprint-tabular` are designed so that they can be called by `~/.../` (that is, `~/pprint-linear/`, `~/pprint-fill/`, and `~/pprint-tabular/`. In particular they take colon and atsign arguments.

As examples of the convenience of specifying pretty printing with `format` strings, consider the functions `pprint-defun` and `pprint-let` used as examples in the last section. They can be more compactly defined as follows. The function `pprint-vector` cannot be defined using `format`, because the data structure it traverses is not a list. The function `pprint-tabular` is inconvenient to define using `format`, because of the need to pass its `tabsize` argument through to a `~:T` directive nested within an iteration over a list.

```
(defun pprint-defun (list)
  (format t "~:<~W ~@_~:I~W ~:_~W~1I ~_~W~:>" list))

(defun pprint-let (list)
  (format t "~:<~W~^ ~:<~@{~:<~@{~W~^ ~_~}~:>~^ ~:_~}~:>~1I~
             ~@{~^ ~_~W~}~:>"
          list))
```

26.5 Compiling Format Control Strings

The control strings used by `format` are essentially programs that perform printing. The macro `formatter` provides the efficiency of using a compiled

function for printing without losing the visual compactness of `format` strings.

[Макрос] **formatter** control-string

The *control-string* must be a literal string. An error is signaled if *control-string* is not a valid `format` control string. The macro `formatter` expands into an expression of the form `(function (lambda (stream &rest args) ...))` that does the printing specified by *control-string*. The `lambda` created accepts an output stream as its first argument and zero or more data values as its remaining arguments. The value returned by the `lambda` is the tail (if any) of the data values that are not printed out by *control-string*. (For example, if the *control-string* is `"~A~A"`, the `cddr` (if any) of the data values is returned.) The form `(formatter "~%~2@{~S, ~}")` is equivalent to the following:

```
#'(lambda (stream &rest args)
    (terpri stream)
    (dotimes (n 2)
      (if (null args) (return nil))
      (prin1 (pop args) stream)
      (write-string ", " stream))
    args)
```

In support of the above mechanism, `format` is extended so that it accepts functions as its second argument as well as strings. When a function is provided, it must be a function of the form created by `formatter`. The function is called with the appropriate output stream as its first argument and the data arguments to `format` as its remaining arguments. The function should perform whatever output is necessary and return the unused tail of the arguments (if any). The directives `~?` and `~{~}` with no body are also extended so that they accept functions as well as control strings. Every other standard function that takes a `format` string as an argument (for example, `error` and `warn`) is also extended so that it can accept functions of the form above instead.

26.6 Pretty Printing Dispatch Tables

When `*print-pretty*` is not `nil`, the pprint dispatch table in the variable `*print-pprint-dispatch*` controls how objects are printed. The information in this table takes precedence over all other mechanisms for specifying how to print objects. In particular, it overrides user-defined `print-object` methods and print functions for structures. However, if there is no specification for how to pretty print a particular kind of object, it is then printed using the standard mechanisms as if `*print-pretty*` were `nil`.

A pprint dispatch table is a mapping from keys to pairs of values. The keys are type specifiers. The values are functions and numerical priorities. Basic insertion and retrieval is done based on the keys with the equality of keys being tested by `equal`. The function to use when pretty printing an object is chosen by finding the highest priority function in `*print-pprint-dispatch*` that is associated with a type specifier that matches the object.

[Function] **copy-pprint-dispatch** **&optional** *table*

A copy is made of *table*, which defaults to the current pprint dispatch table. If *table* is `nil`, a copy is returned of the initial value of `*print-pprint-dispatch*`.

[Function] **pprint-dispatch** *object* **&optional** *table*

This retrieves the highest priority function from a pprint table that is associated with a type specifier in the table that matches *object*. The function is chosen by finding all the type specifiers in *table* that match the object and selecting the highest priority function associated with any of these type specifiers. If there is more than one highest priority function, an arbitrary choice is made. If no type specifiers match the object, a function is returned that prints object with `*print-pretty*` bound to `nil`.

As a second return value, `pprint-dispatch` returns a flag that is `t` if a matching type specifier was found in *table* and `nil` if not.

Table (which defaults to `*print-pprint-dispatch*`) must be a pprint dispatch table. *Table* can be `nil`, in which case retrieval is done in the initial value of `*print-pprint-dispatch*`.

When `*print-pretty*` is `t`, `(write object :stream s)` is equivalent to `(funcall (pprint-dispatch object) s object)`.

[Function] **set-pprint-dispatch** *type function* **&optional** *priority table*

This puts an entry into a pprint dispatch table and returns `nil`. The *type* must be a valid type specifier and is the key of the entry. The first action of `set-pprint-dispatch` is to remove any pre-existing entry associated with *type*. This guarantees that there will never be two entries associated with the same type specifier in a given pprint dispatch table. Equality of type specifiers is tested by `equal`.

Two values are associated with each type specifier in a pprint dispatch table: a function and a priority. The *function* must accept two arguments: the stream to send output to and the object to be printed. The *function* should pretty print the object on the stream. The *function* can assume that object satisfies *type*. The *function* should obey `*print-readably*`. Any values returned by the *function* are ignored.

The *priority* (which defaults to 0) must be a non-complex number. This number is used as a priority to resolve conflicts when an object matches more than one entry. An error is signaled if priority fails to be a non-complex number.

The *table* (which defaults to the value of `*print-pprint-dispatch*`) must be a pprint dispatch table. The specified entry is placed in this table.

It is permissible for *function* to be `nil`. In this situation, there will be no *type* entry in *table* after `set-pprint-dispatch` is evaluated.

To facilitate the use of pprint dispatch tables for controlling the pretty printing of Lisp code, the *type-specifier* argument of the function `set-pprint-dispatch` is allowed to contain the form `(cons` *car-type cdr-type*`)`. This form indicates that the corresponding object must be a cons whose *car* satisfies the type specifier *car-type* and whose *cdr* satisfies the type specifier *cdr-type*. The *cdr-type* can be omitted, in which case it defaults to `t`.

The initial value of `*print-pprint-dispatch*` is implementation-dependent. However, the initial entries all use a special class of priorities that are less than every priority that can be specified using `set-pprint-dispatch`. This guarantees that pretty printing functions specified by users will override everything in the initial value of `*print-pprint-dispatch*`.

Consider the following examples. The first form restores `*print-pprint-dispatch*` to its initial value. The next two forms then specify a special way of pretty printing ratios. Note that the more

specific type specifier has to be associated with a higher priority.

```
(setq *print-pprint-dispatch*
      (copy-pprint-dispatch nil))

(defun div-print (s r colon? atsign?)
  (declare (ignore colon? atsign?))
  (format s "(/ ~D ~D)" (numerator (abs r)) (denominator r)))

(set-pprint-dispatch 'ratio (formatter "#.~/div-print/"))

(set-pprint-dispatch '(and ratio (satisfies minusp))
  (formatter "#.(- ~/div-print/)")
  5)

(pprint '(1/3 -2/3)) prints: (#.(/ 1 3) #.(- (/ 2 3)))
```

The following two forms illustrate the specification of pretty printing functions for particular types of Lisp code. The first form illustrates how to specify the traditional method for printing quoted objects using "'" syntax. Note the care taken to ensure that data lists that happen to begin with `quote` will be printed readably. The second form specifies that lists beginning with the symbol `my-let` should print the same way that lists beginning with `let` print when the initial pprint dispatch table is in effect.

```
(set-pprint-dispatch '(cons (member quote))
  #'(lambda (s list)
      (if (and (consp (cdr list)) (null (cddr list)))
          (funcall (formatter "'~W") s (cadr list))
          (pprint-fill s list)))))

(set-pprint-dispatch '(cons (member my-let))
  (pprint-dispatch '(let) nil))
```

The next example specifies a default method for printing lists that do not correspond to function calls. Note that, as shown in the definition of `pprint-tabular` above, `pprint-linear`, `pprint-fill`, and `pprint-tabular` are defined with optional colon and atsign arguments so that they can be used as pprint dispatch functions as well as `~/.../` functions.

```
(set-pprint-dispatch
  '(cons (not (and symbol (satisfies fboundp))))
  #'pprint-fill
  -5)
```

With a line length of 9, `(pprint '(0 b c d e f g h i j k))` prints:

```
(0 b c d
 e f g h
 i j k)
```

This final example shows how to define a pretty printing function for a user defined data structure.

```
(defstruct family mom kids)

(set-pprint-dispatch 'family
  #'(lambda (s f)
      (format s "~@<#<~;~W and ~2I~_~/pprint-fill/~;>~:>"
              (family-mom f) (family-kids f))))
```

The pretty printing function for the structure `family` specifies how to adjust the layout of the output so that it can fit aesthetically into a variety of line widths. In addition, it obeys the printer control variables `*print-level*`, `*print-length*`, `*print-lines*`, `*print-circle*`, `*print-shared*`, and `*print-escape*`, and can tolerate several different kinds of malformity in the data structure. The output below shows what is printed out with a right margin of 25, `*print-pretty* t`, `*print-escape* nil`, and a malformed `kids` list.

```
(write (list 'principal-family
             (make-family :mom "Lucy"
                          :kids '("Mark" "Bob" . "Dan")))
       :right-margin 25 :pretty T :escape nil :miser-width nil)

(PRINCIPAL-FAMILY
 #<Lucy and
     Mark Bob . Dan>)
```

Note that a pretty printing function for a structure is different from the structure's print function. While print functions are permanently associated with a structure, pretty printing functions are stored in pprint dispatch tables and can be rapidly changed to reflect different printing needs. If there is no pretty printing function for a structure in the current print dispatch table, the print function (if any) is used instead.

Глава 27

Common Lisp Object System

Authors: Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon

X3J13 voted in June 1988 to adopt the first two chapters (of three) of the Common Lisp Object System specification as a part of the forthcoming draft Common Lisp standard.

This chapter presents the bulk of the first two chapters of the Common Lisp Object System specification; it is substantially identical to these two specification chapters as previously published elsewhere [5, 6, 7]. I have edited the material only very lightly to conform to the overall style of this book and to save a substantial number of pages by using a typographically condensed presentation. I have inserted a small number of bracketed remarks, identified by the initials GLS. The chapter divisions of the original specification have become section divisions in this chapter; references to the three chapters of the original specification now refer to the three "parts" of the specification. (See the Acknowledgments to this second edition for acknowledgments to others who contributed to the Common Lisp Object System specification.) This is not the last word on CLOS; X3J13 may well refine this material further. Keene has written a good tutorial introduction to CLOS [26].

—Guy L. Steele Jr.

27.1 Programmer Interface Concepts Концепции интерфейса для программиста

The Объектная система Common Lisp'a (CLOS) is an object-oriented extension to Common Lisp. It is based on generic functions, multiple inheritance, declarative method combination, and a meta-object protocol.

The first two parts of this specification describe the standard Programmer Interface for the Объектная система Common Lisp'a. The first part, Programmer Interface Concepts, contains a description of the concepts of the Объектная система Common Lisp'a, and the second part, Functions in the Programmer Interface, contains a description of the functions and macros in the Объектная система Common Lisp'a Programmer Interface. The third part, The Объектная система Common Lisp'a Meta-Object Protocol, explains how the Объектная система Common Lisp'a can be customized. [The third part has not yet been approved by X3J13 for inclusion in the forthcoming Common Lisp standard and is not included in this book.—GLS]

The fundamental objects of the Объектная система Common Lisp'a are classes, instances, generic functions, and methods.

A *class* object determines the structure and behavior of a set of other objects, which are called its *instances*. Every Common Lisp object is an *instance* of a class. The class of an object determines the set of operations that can be performed on the object.

A *generic function* is a function whose behavior depends on the classes or identities of the arguments supplied to it. A generic function object contains a set of methods, a lambda-list, a method combination type, and other information. The *methods* define the class-specific behavior and operations of the generic function; a method is said to *specialize* a generic function. When invoked, a generic function executes a subset of its methods based on the classes of its arguments.

A generic function can be used in the same ways as an ordinary function in Common Lisp; in particular, a generic function can be used as an argument to `funcall` and `apply` and can be given a global or a local name.

A *method* is an object that contains a method function, a sequence of *parameter specializers* that specify when the given method is applicable, and a sequence of *qualifiers* that is used by the *method combination* facility to distinguish among methods. Each required formal parameter of each method

has an associated parameter specializer, and the method will be invoked only on arguments that satisfy its parameter specializers.

The method combination facility controls the selection of methods, the order in which they are run, and the values that are returned by the generic function. The Объектная система Common Lisp'a offers a default method combination type and provides a facility for declaring new types of method combination.

27.1.1 Error Terminology

The terminology used in this chapter to describe erroneous situations differs from the terminology used in the first edition. The new terminology involves *situations*; a situation is the evaluation of an expression in some specific context. For example, a situation might be the invocation of a function on arguments that fail to satisfy some specified constraints.

In the specification of the Объектная система Common Lisp'a, the behavior of programs in all situations is described, and the options available to the implementor are defined. No implementation is allowed to extend the syntax or semantics of the Объектная система except as explicitly defined in the Объектная система specification. In particular, no implementation is allowed to extend the syntax of the Объектная система in such a way that ambiguity between the specified syntax of the Объектная система and those extensions is possible.

"When situation *S* occurs, an error is signaled." This terminology has the following meaning:

- If this situation occurs, an error will be signaled in the interpreter and in code compiled under all compiler safety optimization levels.
- Valid programs may rely on the fact that an error will be signaled in the interpreter and in code compiled under all compiler safety optimization levels.
- Every implementation is required to detect such an error in the interpreter and in code compiled under all compiler safety optimization levels.

"When situation *S* occurs, an error should be signaled." This terminology has the following meaning:

- If this situation occurs, an error will be signaled at least in the interpreter and in code compiled under the safest compiler safety optimization level.
- Valid programs may not rely on the fact that an error will be signaled.
- Every implementation is required to detect such an error at least in the interpreter and in code compiled under the safest compiler safety optimization level.
- When an error is not signaled, the results are undefined (see below).

"When situation *S* occurs, the results are undefined." This terminology has the following meaning:

- If this situation occurs, the results are unpredictable. The results may range from harmless to fatal.
- Implementations are allowed to detect this situation and signal an error, but no implementation is required to detect the situation.
- No valid program may depend on the effects of this situation, and all valid programs are required to treat the effects of this situation as unpredictable.

"When situation *S* occurs, the results are unspecified." This terminology has the following meaning:

- The effects of this situation are not specified in the Объектная система, but the effects are harmless.
- Implementations are allowed to specify the effects of this situation.
- No portable program can depend on the effects of this situation, and all portable programs are required to treat the situation as unpredictable but harmless.

"The Объектная система Common Lisp'a may be extended to cover situation *S*." The meaning of this terminology is that an implementation is free to treat situation S in one of three ways:

- When situation *S* occurs, an error is signaled at least in the interpreter and in code compiled under the safest compiler safety optimization level.
- When situation *S* occurs, the results are undefined.
- When situation *S* occurs, the results are defined and specified.

In addition, this terminology has the following meaning:

- No portable program can depend on the effects of this situation, and all portable programs are required to treat the situation as undefined.

"Implementations are free to extend the syntax *S*." This terminology has the following meaning:

- Implementations are allowed to define unambiguous extensions to syntax *S*.
- No portable program can depend on this extension, and all portable programs are required to treat the syntax as meaningless.

The Объектная система Common Lisp'a specification may disallow certain extensions while allowing others.

27.1.2 Classes

A *class* is an object that determines the structure and behavior of a set of other objects, which are called its *instances*.

A class can inherit structure and behavior from other classes. A class whose definition refers to other classes for the purpose of inheriting from them is said to be a *subclass* of each of those classes. The classes that are designated for purposes of inheritance are said to be *superclasses* of the inheriting class.

A class can have a *name*. The function `class-name` takes a class object and returns its name. The name of an anonymous class is `nil`. A symbol can *name* a class. The function `find-class` takes a symbol and returns the class that the symbol names. A class has a *proper name* if the name is a symbol and if the name of the class names that class. That is, a class *C* has the

proper name S if $S =$ `(class-name C)` and $C =$ `(find-class S)`. Notice that it is possible for `(find-class` S_1`)` = `(find-class` S_2`)` and $S_1 \neq S_2$. If $C =$ `(find-class S)`, we say that C is the *class named* S.

A class C_1 is a *direct superclass* of a class C_2 if C_2 explicitly designates C_1 as a superclass in its definition. In this case, C_2 is a *direct subclass* of C_1. A class C_n is a *superclass* of a class C_1 if there exists a series of classes $C_2, \ldots, C_{n-1}$ such that C_{i+1} is a direct superclass of C_i for $1 \leq i < n$. In this case, C_1 is a *subclass* of C_n. A class is considered neither a superclass nor a subclass of itself. That is, if C_1 is a superclass of C_2, then $C_1 \neq C_2$. The set of classes consisting of some given class C along with all of its superclasses is called "C and its superclasses."

Each class has a *class precedence list*, which is a total ordering on the set of the given class and its superclasses. The total ordering is expressed as a list ordered from most specific to least specific. The class precedence list is used in several ways. In general, more specific classes can *shadow*, or override, features that would otherwise be inherited from less specific classes. The method selection and combination process uses the class precedence list to order methods from most specific to least specific.

When a class is defined, the order in which its direct superclasses are mentioned in the defining form is important. Each class has a *local precedence order*, which is a list consisting of the class followed by its direct superclasses in the order mentioned in the defining form.

A class precedence list is always consistent with the local precedence order of each class in the list. The classes in each local precedence order appear within the class precedence list in the same order. If the local precedence orders are inconsistent with each other, no class precedence list can be constructed, and an error is signaled. The class precedence list and its computation is discussed in section 27.1.5.

Classes are organized into a *directed acyclic graph*. There are two distinguished classes, named `t` and `standard-object`. The class named `t` has no superclasses. It is a superclass of every class except itself. The class named `standard-object` is an instance of the class `standard-class` and is a superclass of every class that is an instance of `standard-class` except itself.

There is a mapping from the Common Lisp Object System class space into the Common Lisp type space. Many of the standard Common Lisp types have a corresponding class that has the same name as the type. Some Common Lisp types do not have a corresponding class. The integration of

the type and class systems is discussed in section 27.1.4.

Classes are represented by objects that are themselves instances of classes. The class of the class of an object is termed the *metaclass* of that object. When no misinterpretation is possible, the term *metaclass* will be used to refer to a class that has instances that are themselves classes. The metaclass determines the form of inheritance used by the classes that are its instances and the representation of the instances of those classes. The Объектная система Common Lisp'a provides a default metaclass, `standard-class`, that is appropriate for most programs. The meta-object protocol provides mechanisms for defining and using new metaclasses.

Except where otherwise specified, all classes mentioned in this chapter are instances of the class `standard-class`, all generic functions are instances of the class `standard-generic-function`, and all methods are instances of the class `standard-method`.

Defining Classes

The macro `defclass` is used to define a new named class. The definition of a class includes the following:

- The name of the new class. For newly defined classes this is a proper name.
- The list of the direct superclasses of the new class.
- A set of *slot specifiers*. Each slot specifier includes the name of the slot and zero or more *slot options*. A slot option pertains only to a single slot. If a class definition contains two slot specifiers with the same name, an error is signaled.
- A set of *class options*. Each class option pertains to the class as a whole.

The slot options and class options of the `defclass` form provide mechanisms for the following:

- Supplying a default initial value form for a given slot.
- Requesting that methods for generic functions be automatically generated for reading or writing slots.

- Controlling whether a given slot is shared by instances of the class or whether each instance of the class has its own slot.
- Supplying a set of initialization arguments and initialization argument defaults to be used in instance creation.
- Indicating that the metaclass is to be other than the default.
- Indicating the expected type for the value stored in the slot.
- Indicating the documentation string for the slot.

Creating Instances of Classes

The generic function `make-instance` creates and returns a new instance of a class. The Объектная система provides several mechanisms for specifying how a new instance is to be initialized. For example, it is possible to specify the initial values for slots in newly created instances either by giving arguments to `make-instance` or by providing default initial values.

Further initialization activities can be performed by methods written for generic functions that are part of the initialization protocol. The complete initialization protocol is described in section 27.1.9.

Slots

An object that has `standard-class` as its metaclass has zero or more named slots. The slots of an object are determined by the class of the object. Each slot can hold one value. The name of a slot is a symbol that is syntactically valid for use as a variable name.

When a slot does not have a value, the slot is said to be *unbound*. When an unbound slot is read, the generic function `slot-unbound` is invoked. The system-supplied primary method for `slot-unbound` signals an error.

The default initial value form for a slot is defined by the `:initform` slot option. When the `:initform` form is used to supply a value, it is evaluated in the lexical environment in which the `defclass` form was evaluated. The `:initform` along with the lexical environment in which the `defclass` form was evaluated is called a *captured* `:initform`. See section 27.1.9.

A *local slot* is defined to be a slot that is visible to exactly one instance, namely the one in which the slot is allocated. A *shared slot* is defined to

be a slot that is visible to more than one instance of a given class and its subclasses.

A class is said to *define* a slot with a given name when the `defclass` form for that class contains a slot specifier with that name. Defining a local slot does not immediately create a slot; it causes a slot to be created each time an instance of the class is created. Defining a shared slot immediately creates a slot.

The `:allocation` slot option to `defclass` controls the kind of slot that is defined. If the value of the `:allocation` slot option is `:instance`, a local slot is created. If the value of `:allocation` is `:class`, a shared slot is created.

A slot is said to be *accessible* in an instance of a class if the slot is defined by the class of the instance or is inherited from a superclass of that class.

At most one slot of a given name can be accessible in an instance. A shared slot defined by a class is accessible in all instances of that class. A detailed explanation of the inheritance of slots is given in section 27.1.3.

Accessing Slots

Slots can be accessed in two ways: by use of the primitive function `slot-value` and by use of generic functions generated by the `defclass` form.

The function `slot-value` can be used with any slot name specified in the `defclass` form to access a specific slot accessible in an instance of the given class.

The macro `defclass` provides syntax for generating methods to read and write slots. If a *reader* is requested, a method is automatically generated for reading the value of the slot, but no method for storing a value into it is generated. If a *writer* is requested, a method is automatically generated for storing a value into the slot, but no method for reading its value is generated. If an *accessor* is requested, a method for reading the value of the slot and a method for storing a value into the slot are automatically generated. Reader and writer methods are implemented using `slot-value`.

When a reader or writer is specified for a slot, the name of the generic function to which the generated method belongs is directly specified. If the name specified for the writer option is the symbol *name*, the name of the generic function for writing the slot is the symbol *name*, and the generic function takes two arguments: the new value and the instance, in that order. If the name specified for the accessor option is the symbol *name*, the name of the generic function for reading the slot is the symbol *name*, and the name of the generic function for writing the slot is the list `(setf` *name*`)`.

A generic function created or modified by supplying reader, writer, or accessor slot options can be treated exactly as an ordinary generic function.

Note that `slot-value` can be used to read or write the value of a slot whether or not reader or writer methods exist for that slot. When `slot-value` is used, no reader or writer methods are invoked.

The macro `with-slots` can be used to establish a lexical environment in which specified slots are lexically available as if they were variables. The macro `with-slots` invokes the function `slot-value` to access the specified slots.

The macro `with-accessors` can be used to establish a lexical environment in which specified slots are lexically available through their accessors

as if they were variables. The macro `with-accessors` invokes the appropriate accessors to access the specified slots. Any accessors specified by `with-accessors` must already have been defined before they are used.

27.1.3 Inheritance

A class can inherit methods, slots, and some `defclass` options from its superclasses. The following sections describe the inheritance of methods, the inheritance of slots and slot options, and the inheritance of class options.

Inheritance of Methods

A subclass inherits methods in the sense that any method applicable to all instances of a class is also applicable to all instances of any subclass of that class.

The inheritance of methods acts the same way regardless of whether the method was created by using one of the method-defining forms or by using one of the `defclass` options that causes methods to be generated automatically.

The inheritance of methods is described in detail in section 27.1.7.

Inheritance of Slots and Slot Options

The set of names of all slots accessible in an instance of a class C is the union of the sets of names of slots defined by C and its superclasses. The *structure* of an instance is the set of names of local slots in that instance.

In the simplest case, only one class among C and its superclasses defines a slot with a given slot name. If a slot is defined by a superclass of C, the slot is said to be *inherited*. The characteristics of the slot are determined by the slot specifier of the defining class. Consider the defining class for a slot S. If the value of the `:allocation` slot option is `:instance`, then S is a local slot and each instance of C has its own slot named S that stores its own value. If the value of the `:allocation` slot option is `:class`, then S is a shared slot, the class that defined S stores the value, and all instances of C can access that single slot. If the `:allocation` slot option is omitted, `:instance` is used.

In general, more than one class among C and its superclasses can define a slot with a given name. In such cases, only one slot with the given name is accessible in an instance of C, and the characteristics of that slot are a combination of the several slot specifiers, computed as follows:

- All the slot specifiers for a given slot name are ordered from most specific to least specific, according to the order in C's class precedence

list of the classes that define them. All references to the specificity of slot specifiers immediately following refer to this ordering.

- The allocation of a slot is controlled by the most specific slot specifier. If the most specific slot specifier does not contain an `:allocation` slot option, `:instance` is used. Less specific slot specifiers do not affect the allocation.

- The default initial value form for a slot is the value of the `:initform` slot option in the most specific slot specifier that contains one. If no slot specifier contains an `:initform` slot option, the slot has no default initial value form.

- The contents of a slot will always be of type `(and` T_1 ... T_n`)` where $T_1, \ldots, T_n$ are the values of the `:type` slot options contained in all of the slot specifiers. If no slot specifier contains the `:type` slot option, the contents of the slot will always be of type `t`. The result of attempting to store in a slot a value that does not satisfy the type of the slot is undefined.

- The set of initialization arguments that initialize a given slot is the union of the initialization arguments declared in the `:initarg` slot options in all the slot specifiers.

- The documentation string for a slot is the value of the `:documentation` slot option in the most specific slot specifier that contains one. If no slot specifier contains a `:documentation` slot option, the slot has no documentation string.

A consequence of the allocation rule is that a shared slot can be shadowed. For example, if a class C_1 defines a slot named S whose value for the `:allocation` slot option is `:class`, that slot is accessible in instances of C_1 and all of its subclasses. However, if C_2 is a subclass of C_1 and also defines a slot named S, C_1's slot is not shared by instances of C_2 and its subclasses. When a class C_1 defines a shared slot, any subclass C_2 of C_1 will share this single slot unless the `defclass` form for C_2 specifies a slot of the same name or there is a superclass of C_2 that precedes C_1 in the class precedence list of C_2 that defines a slot of the same name.

A consequence of the type rule is that the value of a slot satisfies the type constraint of each slot specifier that contributes to that slot. Because the

result of attempting to store in a slot a value that does not satisfy the type constraint for the slot is undefined, the value in a slot might fail to satisfy its type constraint.

The `:reader`, `:writer`, and `:accessor` slot options create methods rather than define the characteristics of a slot. Reader and writer methods are inherited in the sense described in section 27.1.3.

Methods that access slots use only the name of the slot and the type of the slot's value. Suppose a superclass provides a method that expects to access a shared slot of a given name, and a subclass defines a local slot with the same name. If the method provided by the superclass is used on an instance of the subclass, the method accesses the local slot.

Inheritance of Class Options

The `:default-initargs` class option is inherited. The set of defaulted initialization arguments for a class is the union of the sets of initialization arguments specified in the `:default-initargs` class options of the class and its superclasses. When more than one default initial value form is supplied for a given initialization argument, the default initial value form that is used is the one supplied by the class that is most specific according to the class precedence list.

If a given `:default-initargs` class option specifies an initialization argument of the same name more than once, an error is signaled.

Examples

```
(defclass C1 ()
  ((S1 :initform 5.4 :type number)
   (S2 :allocation :class)))

(defclass C2 (C1)
  ((S1 :initform 5 :type integer)
   (S2 :allocation :instance)
   (S3 :accessor C2-S3)))
```

Instances of the class `C1` have a local slot named `S1`, whose default initial value is 5.4 and whose value should always be a number. The class `C1` also has a shared slot named `S2`.

There is a local slot named `S1` in instances of `C2`. The default initial value of `S1` is 5. The value of `S1` will be of type `(and integer number)`. There are also local slots named `S2` and `S3` in instances of `C2`. The class `C2` has a method for `C2-S3` for reading the value of slot `S3`; there is also a method for `(setf C2-S3)` that writes the value of `S3`.

27.1.4 Integrating Types and Classes

The Объектная система Common Lisp'а maps the space of classes into the Common Lisp type space. Every class that has a proper name has a corresponding type with the same name.

The proper name of every class is a valid type specifier. In addition, every class object is a valid type specifier. Thus the expression `(typep` *`object class`*`)` evaluates to true if the class of *object* is *class* itself or a subclass of *class*. The evaluation of the expression `(subtypep` *`class1 class2`*`)` returns the values `t` and `t` if *class1* is a subclass of *class2* or if they are the same class; otherwise it returns the values `nil` and `t`. If I is an instance of some class C named S and C is an instance of `standard-class`, the evaluation of the expression `(type-of` *`I`*`)` will return S if S is the proper name of C; if S is not the proper name of C, the expression `(type-of` *`I`*`)` will return C.

Because the names of classes and class objects are type specifiers, they may be used in the special operator `the` and in type declarations.

Many but not all of the predefined Common Lisp type specifiers have a corresponding class with the same proper name as the type. These type specifiers are listed in table 27.1. For example, the type `array` has a corresponding class named `array`. No type specifier that is a list, such as `(vector double-float 100)`, has a corresponding class. The form `deftype` does not create any classes.

Each class that corresponds to a predefined Common Lisp type specifier can be implemented in one of three ways, at the discretion of each implementation. It can be a *standard class* (of the kind defined by `defclass`), a *structure class* (defined by `defstruct`), or a *built-in class* (implemented in a special, non-extensible way).

A built-in class is one whose instances have restricted capabilities or special representations. Attempting to use `defclass` to define subclasses of a built-in class signals an error. Calling `make-instance` to create an instance of a built-in class signals an error. Calling `slot-value` on an instance of a built-

in class signals an error. Redefining a built-in class or using `change-class` to change the class of an instance to or from a built-in class signals an error. However, built-in classes can be used as parameter specializers in methods.

It is possible to determine whether a class is a built-in class by checking the metaclass. A standard class is an instance of `standard-class`, a built-in class is an instance of `built-in-class`, and a structure class is an instance of `structure-class`.

Each structure type created by `defstruct` without using the `:type` option has a corresponding class. This class is an instance of `structure-class`. The `:include` option of `defstruct` creates a direct subclass of the class that corresponds to the included structure.

The purpose of specifying that many of the standard Common Lisp type specifiers have a corresponding class is to enable users to write methods that discriminate on these types. Method selection requires that a class precedence list can be determined for each class.

The hierarchical relationships among the Common Lisp type specifiers are mirrored by relationships among the classes corresponding to those types. The existing type hierarchy is used for determining the class precedence list for each class that corresponds to a predefined Common Lisp type. In some cases, the first edition did not specify a local precedence order for two supertypes of a given type specifier. For example, `null` is a subtype of both `symbol` and `list`, but the first edition did not specify whether `symbol` is more specific or less specific than `list`. The CLOS specification defines those relationships for all such classes.

Table 27.1 lists the set of classes required by the Объектная система that correspond to predefined Common Lisp type specifiers. The superclasses of each such class are presented in order from most specific to most general, thereby defining the class precedence list for the class. The local precedence order for each class that corresponds to a Common Lisp type specifier can be derived from this table.

Individual implementations may be extended to define other type specifiers to have a corresponding class. Individual implementations can be extended to add other subclass relationships and to add other elements to the class precedence lists in the above table as long as they do not violate the type relationships and disjointness requirements specified in section 2.15. A standard class defined with no direct superclasses is guaranteed to be disjoint from all of the classes in the table, except for the class named `t`.

[At this point the original CLOS report specified that certain Common

Lisp types were to appear in table 27.1 if and only if X3J13 voted to make them disjoint from `cons`, `symbol`, `array`, `number`, and `character`. X3J13 voted to do so in June 1988 . I have added these types and their class precedence lists to the table; the new types are indicated by asterisks.—GLS]

27.1.5 Determining the Class Precedence List

The `defclass` form for a class provides a total ordering on that class and its direct superclasses. This ordering is called the *local precedence order*. It is an ordered list of the class and its direct superclasses. The *class precedence list* for a class C is a total ordering on C and its superclasses that is consistent with the local precedence orders for C and its superclasses.

A class precedes its direct superclasses, and a direct superclass precedes all other direct superclasses specified to its right in the superclasses list of the `defclass` form. For every class C, define

$$R_C = \{(C, C_1), (C_1, C_2), \ldots, (C_{n-1}, C_n)\}$$

where $C_1, \ldots, C_n$ are the direct superclasses of C in the order in which they are mentioned in the `defclass` form. These ordered pairs generate the total ordering on the class C and its direct superclasses.

Let S_C be the set of C and its superclasses. Let R be

$$R = \bigcup_{c \in S_C} R_c$$

The set R may or may not generate a partial ordering, depending on whether the R_c, $c \in S_C$, are consistent; it is assumed that they are consistent and that R generates a partial ordering. When the R_c are not consistent, it is said that R is inconsistent.

To compute the class precedence list for C, topologically sort the elements of S_C with respect to the partial ordering generated by R. When the topological sort must select a class from a set of two or more classes, none of which are preceded by other classes with respect to R, the class selected is chosen deterministically, as described below. If R is inconsistent, an error is signaled.

Topological Sorting

Topological sorting proceeds by finding a class C in S_C such that no other class precedes that element according to the elements in R. The class C is placed first in the result. Remove C from S_C, and remove all pairs of the form (C, D), $D \in S_C$, from R. Repeat the process, adding classes with no predecessors to the end of the result. Stop when no element can be found that has no predecessor.

If S_C is not empty and the process has stopped, the set R is inconsistent. If every class in the finite set of classes is preceded by another, then R contains a loop. That is, there is a chain of classes $C_1, \ldots, C_n$ such that C_i precedes C_{i+1}, $1 \leq i < n$, and C_n precedes C_1.

Sometimes there are several classes from S_C with no predecessors. In this case select the one that has a direct subclass rightmost in the class precedence list computed so far. If there is no such candidate class, R does not generate a partial ordering—the R_c, $c \in S_C$, are inconsistent.

In more precise terms, let $\{N_1, \ldots, N_m\}$, $m \geq 2$, be the classes from S_C with no predecessors. Let $(C_1 \ldots C_n)$, $n \geq 1$, be the class precedence list constructed so far. C_1 is the most specific class, and C_n is the least specific. Let $1 \leq j \leq n$ be the largest number such that there exists an i where $1 \leq i \leq m$ and N_i is a direct superclass of C_j; N_i is placed next.

The effect of this rule for selecting from a set of classes with no predecessors is that classes in a simple superclass chain are adjacent in the class precedence list and that classes in each relatively separated subgraph are adjacent in the class precedence list. For example, let T_1 and T_2 be subgraphs whose only element in common is the class J. Suppose that no superclass of J appears in either T_1 or T_2. Let C_1 be the bottom of T_1; and let C_2 be the bottom of T_2. Suppose C is a class whose direct superclasses are C_1 and C_2 in that order; then the class precedence list for C will start with C and will be followed by all classes in T_1 except J. All the classes of T_2 will be next. The class J and its superclasses will appear last.

Examples

This example determines a class precedence list for the class `pie`. The following classes are defined:

```
(defclass pie (apple cinnamon) ())
(defclass apple (fruit) ())
(defclass cinnamon (spice) ())
(defclass fruit (food) ())
(defclass spice (food) ())
(defclass food () ())
```

The set S={`pie`, `apple`, `cinnamon`, `fruit`, `spice`, `food`, `standard-object`, `t`}. The set R={(`pie`, `apple`), (`apple`, `cinnamon`), (`cinnamon`, `standard-object`), (`apple`, `fruit`), (`fruit`, `standard-object`), (`cinnamon`, `spice`), (`spice`, `standard-object`), (`fruit`, `food`), (`food`, `standard-object`), (`spice`, `food`), (`standard-object`, `t`)}.
[The original CLOS specification [5, 6] contained a minor error in this example: the pairs (`cinnamon`, `standard-object`), (`fruit`, `standard-object`), and (`spice`, `standard-object`) were inadvertently omitted from R in the preceding paragraph. It is important to understand that `defclass` implicitly appends the class `standard-object` to the list of superclasses when the metaclass is `standard-class` (the normal situation), in order to insure that `standard-object` will be a superclass of every instance of `standard-class` except `standard-object` itself (see section 27.1.2). R_c is then generated from this augmented list of superclasses; this is where the extra pairs come from. I have corrected the example by adding these pairs as appropriate throughout the example. The final result, the class precedence list for `pie`, is unchanged.—GLS]
The class `pie` is not preceded by anything, so it comes first; the result so far is `(pie)`. Remove `pie` from S and pairs mentioning `pie` from R to get S={`apple`, `cinnamon`, `fruit`, `spice`, `food`, `standard-object`, `t`} and R={(`apple`, `cinnamon`), (`cinnamon`, `standard-object`), (`apple`, `fruit`), (`fruit`, `standard-object`), (`cinnamon`, `spice`), (`spice`, `standard-object`), (`fruit`, `food`), (`food`, `standard-object`), (`spice`, `food`), (`standard-object`, `t`)}.

The class `apple` is not preceded by anything, so it is next; the result is `(pie apple)`. Removing `apple` and the relevant pairs results in S={`cinnamon`, `fruit`, `spice`, `food`, `standard-object`, `t`} and R={(`cinnamon`, `standard-object`), (`fruit`, `standard-object`), (`cinnamon`, `spice`), (`spice`, `standard-object`), (`fruit`, `food`), (`food`, `standard-object`), (`spice`, `food`), (`standard-object`, `t`)}.
The classes `cinnamon` and `fruit` are not preceded by anything, so the one with a direct subclass rightmost in the class precedence list computed so far goes next. The class `apple` is a direct subclass of `fruit`, and the class `pie` is a direct subclass of `cinnamon`. Because `apple` appears to the right of `pie` in the precedence list, `fruit` goes next, and the result so far is `(pie apple fruit)`. S={`cinnamon`, `spice`, `food`, `standard-object`, `t`}; R={(`cinnamon`, `standard-object`), (`cinnamon`, `spice`), (`spice`, `standard-object`), (`food`, `standard-object`), (`spice`, `food`), (`standard-object`, `t`)}.
The class `cinnamon` is next, giving the result so far as `(pie apple fruit cinnamon)`. At this point S={`spice`, `food`, `standard-object`, `t`}; R={(`spice`, `standard-object`), (`food`, `standard-object`), (`spice`, `food`), (`standard-object`, `t`)}.

The classes `spice`, `food`, `standard-object`, and `t` are then added in that order, and the final class precedence list for `pie` is

(pie apple fruit cinnamon spice food standard-object t)

It is possible to write a set of class definitions that cannot be ordered. For example:

```
(defclass new-class (fruit apple) ())
(defclass apple (fruit) ())
```

The class `fruit` must precede `apple` because the local ordering of superclasses must be preserved. The class `apple` must precede `fruit` because a class always precedes its own superclasses. When this situation occurs, an error is signaled when the system tries to compute the class precedence list.

The following might appear to be a conflicting set of definitions:

```
(defclass pie (apple cinnamon) ())
(defclass pastry (cinnamon apple) ())
(defclass apple () ())
(defclass cinnamon () ())
```

The class precedence list for `pie` is

```
(pie apple cinnamon standard-object t)
```

The class precedence list for `pastry` is

```
(pastry cinnamon apple standard-object t)
```

It is not a problem for `apple` to precede `cinnamon` in the ordering of the superclasses of `pie` but not in the ordering for `pastry`. However, it is not possible to build a new class that has both `pie` and `pastry` as superclasses.

27.1.6 Generic Functions and Methods

A *generic function* is a function whose behavior depends on the classes or identities of the arguments supplied to it. The *methods* define the class-specific behavior and operations of the generic function. The following sections describe generic functions and methods.

Introduction to Generic Functions

A generic function object contains a set of methods, a lambda-list, a method combination type, and other information.

Like an ordinary Lisp function, a generic function takes arguments, performs a series of operations, and perhaps returns useful values. An ordinary function has a single body of code that is always executed when the function is called. A generic function has a set of bodies of code of which a subset is selected for execution. The selected bodies of code and the manner of their combination are determined by the classes or identities of one or more of the arguments to the generic function and by its method combination type.

Ordinary functions and generic functions are called with identical function-call syntax.

Generic functions are true functions that can be passed as arguments, returned as values, used as the first argument to `funcall` and `apply`, and otherwise used in all the ways an ordinary function may be used.

A name can be given to an ordinary function in one of two ways: a *global* name can be given to a function using the `defun` construct; a *local* name can be given using the `flet` or `labels` special operators. A generic function can be given a global name using the `defmethod` or `defgeneric` construct. A generic function can be given a local name using the `generic-flet`, `generic-labels`, or `with-added-methods` special operators. The name of a generic function, like the name of an ordinary function, can be either a symbol or a two-element list whose first element is `setf` and whose second element is a symbol. This is true for both local and global names.

The `generic-flet` special operator creates new local generic functions using the set of methods specified by the method definitions in the `generic-flet` form. The scoping of generic function names within a `generic-flet` form is the same as for `flet`.

The `generic-labels` special operator creates a set of new mutually recursive local generic functions using the set of methods specified by the method definitions in the `generic-labels` form. The scoping of generic function names within a `generic-labels` form is the same as for `labels`.

The `with-added-methods` special operator creates new local generic functions by adding the set of methods specified by the method definitions with a given name in the `with-added-methods` form to copies of the methods of the lexically visible generic function of the same name. If there is a lexically visible ordinary function of the same name as one of the specified generic functions, that function becomes the method function of the default method for the new generic function of that name.

The `generic-function` macro creates an anonymous generic function with the set of methods specified by the method definitions that appear in the `generic-function` form.

When a `defgeneric` form is evaluated, one of three actions is taken:

- If a generic function of the given name already exists, the existing generic function object is modified. Methods specified by the current `defgeneric` form are added, and any methods in the existing generic

function that were defined by a previous `defgeneric` form are removed. Methods added by the current `defgeneric` form might replace methods defined by `defmethod` or `defclass`. No other methods in the generic function are affected or replaced.

- If the given name names a non-generic function, a macro, or a special operator, an error is signaled.
- Otherwise a generic function is created with the methods specified by the method definitions in the `defgeneric` form.

Some forms specify the options of a generic function, such as the type of method combination it uses or its argument precedence order. They will be referred to as "forms that specify generic function options." These forms are `defgeneric`, `generic-function`, `generic-flet`, `generic-labels`, and `with-added-methods`.

Some forms define methods for a generic function. They will be referred to as "method-defining forms." These forms are `defgeneric`, `defmethod`, `generic-function`, `generic-flet`, `generic-labels`, `with-added-methods`, and `defclass`. Note that all the method-defining forms except `defclass` and `defmethod` are also forms that specify generic function options.

Introduction to Methods

A method object contains a method function, a sequence of *parameter specializers* that specify when the given method is applicable, a lambda-list, and a sequence of *qualifiers* that are used by the method combination facility to distinguish among methods.

A method object is not a function and cannot be invoked as a function. Various mechanisms in the Объектная система take a method object and invoke its method function, as is the case when a generic function is invoked. When this occurs it is said that the method is invoked or called.

A method-defining form contains the code that is to be run when the arguments to the generic function cause the method that it defines to be invoked. When a method-defining form is evaluated, a method object is created and one of four actions is taken:

- If a generic function of the given name already exists and if a method object already exists that agrees with the new one on parameter specializers and qualifiers, the new method object replaces the old one. For a definition of one method agreeing with another on parameter specializers and qualifiers, see section 27.1.6.

- If a generic function of the given name already exists and if there is no method object that agrees with the new one on parameter specializers and qualifiers, the existing generic function object is modified to contain the new method object.

- If the given name names a non-generic function, a macro, or a special operator, an error is signaled.

- Otherwise a generic function is created with the methods specified by the method-defining form.

If the lambda-list of a new method is not congruent with the lambda-list of the generic function, an error is signaled. If a method-defining form that cannot specify generic function options creates a new generic function, a lambda-list for that generic function is derived from the lambda-lists of the methods in the method-defining form in such a way as to be congruent with them. For a discussion of *congruence*, see section 27.1.6.

Each method has a *specialized lambda-list*, which determines when that method can be applied. A specialized lambda-list is like an ordinary lambda-list except that a *specialized parameter* may occur instead of the name of a required parameter. A specialized parameter is a list `(`*`variable-name parameter-specializer-name`*`)`, where *parameter-specializer-name* is either a name that names a class or a list `(eql` *`form`*`)`. A parameter specializer name denotes a parameter specializer as follows:

- A name that names a class denotes that class.

- The list `(eql` *`form`*`)` denotes the type specifier `(eql` *`object`*`)`, where *object* is the result of evaluating *form*. The form *form* is evaluated in the lexical environment in which the method-defining form is evaluated. Note that *form* is evaluated only once, at the time the method is defined, not each time the generic function is called.

Parameter specializer names are used in macros intended as the user-level interface (`defmethod`), while parameter specializers are used in the functional interface.

[It is very important to understand clearly the distinction made in the preceding paragraph. A parameter specializer name has the form of a type specifier but is semantically quite different from a type specifier: a parameter specializer name of the form `(eql` *`form`*`)` is not a type specifier, for it contains a *form* to be evaluated. Type specifiers never contain forms to be evaluated. All parameter specializers (as opposed to parameter specializer names) are valid type specifiers, but not all type specifiers are valid parameter specializers. Macros such as `defmethod` take parameter specializer names and treat them as specifications for constructing certain type specifiers (parameter specializers) that may then be used with such functions as `find-method`.—GLS]

Only required parameters may be specialized, and there must be a parameter specializer for each required parameter. For notational simplicity, if some required parameter in a specialized lambda-list in a method-defining form is simply a variable name, its parameter specializer defaults to the class named `t`.

Given a generic function and a set of arguments, an *applicable method* is a method for that generic function whose parameter specializers are satisfied by their corresponding arguments. The following definition specifies what it means for a method to be applicable and for an argument to satisfy a parameter specializer.

Let $\langle A_1, \ldots, A_n \rangle$ be the required arguments to a generic function in order. Let $\langle P_1, \ldots, P_n \rangle$ be the parameter specializers corresponding to the required parameters of the method M in order. The method M is *applicable* when each A_i *satisfies* P_i. If P_i is a class, and if A_i is an instance of a class C, then it is said that A_i *satisfies* P_i when $C = P_i$ or when C is a subclass of P_i. If P_i is of the form `(eql` *`object`*`)`, then it is said that A_i satisfies P_i when the function `eql` applied to A_i and *object* is true.

Because a parameter specializer is a type specifier, the function `typep` can be used during method selection to determine whether an argument satisfies a parameter specializer. In general a parameter specializer cannot be a type specifier list, such as `(vector single-float)`. The only parameter specializer that can be a list is `(eql` *`object`*`)`. This requires that Common Lisp define the type specifier `eql` as if the following were evaluated:

(deftype eql (*object*) '(member ,*object*))

[See section 4.3.—GLS]

A method all of whose parameter specializers are the class named `t` is called a *default method*; it is always applicable but may be shadowed by a more specific method.

Methods can have *qualifiers*, which give the method combination procedure a way to distinguish among methods. A method that has one or more qualifiers is called a *qualified* method. A method with no qualifiers is called an *unqualified method*. A qualifier is any object other than a list, that is, any non-`nil` atom. The qualifiers defined by standard method combination and by the built-in method combination types are symbols.

In this specification, the terms *primary method* and *auxiliary method* are used to partition methods within a method combination type according to their intended use. In standard method combination, primary methods are unqualified methods, and auxiliary methods are methods with a single qualifier that is one of `:around`, `:before`, or `:after`. When a method combination type is defined using the short form of `define-method-combination`, primary methods are methods qualified with the name of the type of method combination, and auxiliary methods have the qualifier `:around`. Thus the terms *primary method* and *auxiliary method* have only a relative definition within a given method combination type.

Agreement on Parameter Specializers and Qualifiers

Two methods are said to agree with each other on parameter specializers and qualifiers if the following conditions hold:

- Both methods have the same number of required parameters. Suppose the parameter specializers of the two methods are $P_{1,1} \ldots P_{1,n}$ and $P_{2,1} \ldots P_{2,n}$.
- For each $1 \leq i \leq n$, $P_{1,i}$ agrees with $P_{2,i}$. The parameter specializer $P_{1,i}$ agrees with $P_{2,i}$ if $P_{1,i}$ and $P_{2,i}$ are the same class or if $P_{1,i}$ = `(eql` *object*$_1$`)`, $P_{2,i}$ = `(eql` *object*$_2$`)`, and `(eql` *object*$_1$ *object*$_2$`)`. Otherwise $P_{1,i}$ and $P_{2,i}$ do not agree.

- The lists of qualifiers of both methods contain the same non-`nil` atoms in the same order. That is, the lists are `equal`.

Congruent Lambda-Lists for All Methods of a Generic Function

These rules define the congruence of a set of lambda-lists, including the lambda-list of each method for a given generic function and the lambda-list specified for the generic function itself, if given.

- Each lambda-list must have the same number of required parameters.
- Each lambda-list must have the same number of optional parameters. Each method can supply its own default for an optional parameter.
- If any lambda-list mentions `&rest` or `&key`, each lambda-list must mention one or both of them.
- If the generic function lambda-list mentions `&key`, each method must accept all of the keyword names mentioned after `&key`, either by accepting them explicitly, by specifying `&allow-other-keys`, or by specifying `&rest` but not `&key`. Each method can accept additional keyword arguments of its own. The checking of the validity of keyword names is done in the generic function, not in each method. A method is invoked as if the keyword argument pair whose keyword is `:allow-other-keys` and whose value is `t` were supplied, though no such argument pair will be passed.
- The use of `&allow-other-keys` need not be consistent across lambda-lists. If `&allow-other-keys` is mentioned in the lambda-list of any applicable method or of the generic function, any keyword arguments may be mentioned in the call to the generic function.
- The use of `&aux` need not be consistent across methods.

If a method-defining form that cannot specify generic function options creates a generic function, and if the lambda-list for the method mentions keyword arguments, the lambda-list of the generic function will mention `&key` (but no keyword arguments).

Keyword Arguments in Generic Functions and Methods

When a generic function or any of its methods mentions `&key` in a lambda-list, the specific set of keyword arguments accepted by the generic function varies according to the applicable methods. The set of keyword arguments accepted by the generic function for a particular call is the union of the keyword arguments accepted by all applicable methods and the keyword arguments mentioned after `&key` in the generic function definition, if any. A method that has `&rest` but not `&key` does not affect the set of acceptable keyword arguments. If the lambda-list of any applicable method or of the generic function definition contains `&allow-other-keys`, all keyword arguments are accepted by the generic function.

The lambda-list congruence rules require that each method accept all of the keyword arguments mentioned after `&key` in the generic function definition, by accepting them explicitly, by specifying `&allow-other-keys`, or by specifying `&rest` but not `&key`. Each method can accept additional keyword arguments of its own, in addition to the keyword arguments mentioned in the generic function definition.

If a generic function is passed a keyword argument that no applicable method accepts, an error is signaled.

For example, suppose there are two methods defined for `width` as follows:

```
(defmethod width ((c character-class) &key font) ...)

(defmethod width ((p picture-class) &key pixel-size) ...)
```

Assume that there are no other methods and no generic function definition for `width`. The evaluation of the following form will signal an error because the keyword argument `:pixel-size` is not accepted by the applicable method.

```
(width (make-instance 'character-class :char #\Q)
       :font 'baskerville :pixel-size 10)
```

The evaluation of the following form will signal an error.

```
(width (make-instance 'picture-class :glyph (glyph #\Q))
       :font 'baskerville :pixel-size 10)
```

The evaluation of the following form will not signal an error if the class named `character-picture-class` is a subclass of both `picture-class` and `character-class`.

```
(width (make-instance 'character-picture-class :char #\Q)
       :font 'baskerville :pixel-size 10)
```

27.1.7 Method Selection and Combination

When a generic function is called with particular arguments, it must determine the code to execute. This code is called the *effective method* for those arguments. The effective method is a *combination* of the applicable methods in the generic function. A combination of methods is a Lisp expression that contains calls to some or all of the methods. If a generic function is called and no methods apply, the generic function `no-applicable-method` is invoked.

When the effective method has been determined, it is invoked with the same arguments that were passed to the generic function. Whatever values it returns are returned as the values of the generic function.

Determining the Effective Method

The effective method for a set of arguments is determined by the following three-step procedure:

1. Select the applicable methods.
2. Sort the applicable methods by precedence order, putting the most specific method first.
3. Apply method combination to the sorted list of applicable methods, producing the effective method.

Selecting the Applicable Methods. This step is described in section 27.1.6.

Sorting the Applicable Methods by Precedence Order. To compare the precedence of two methods, their parameter specializers are examined in order. The default examination order is from left to right, but an alternative order may be specified by the `:argument-precedence-order` option to `defgeneric` or to any of the other forms that specify generic function options.

The corresponding parameter specializers from each method are compared. When a pair of parameter specializers are equal, the next pair are compared for equality. If all corresponding parameter specializers are equal,

the two methods must have different qualifiers; in this case, either method can be selected to precede the other.

If some corresponding parameter specializers are not equal, the first pair of parameter specializers that are not equal determines the precedence. If both parameter specializers are classes, the more specific of the two methods is the method whose parameter specializer appears earlier in the class precedence list of the corresponding argument. Because of the way in which the set of applicable methods is chosen, the parameter specializers are guaranteed to be present in the class precedence list of the class of the argument.

If just one parameter specializer is `(eql` *object*`)`, the method with that parameter specializer precedes the other method. If both parameter specializers are `eql` forms, the specializers must be the same (otherwise the two methods would not both have been applicable to this argument).

The resulting list of applicable methods has the most specific method first and the least specific method last.

Applying Method Combination to the Sorted List of Applicable Methods. In the simple case—if standard method combination is used and all applicable methods are primary methods—the effective method is the most specific method. That method can call the next most specific method by using the function `call-next-method`. The method that `call-next-method` will call is referred to as the *next method*. The predicate `next-method-p` tests whether a next method exists. If `call-next-method` is called and there is no next most specific method, the generic function `no-next-method` is invoked.

In general, the effective method is some combination of the applicable methods. It is defined by a Lisp form that contains calls to some or all of the applicable methods, returns the value or values that will be returned as the value or values of the generic function, and optionally makes some of the methods accessible by means of `call-next-method`. This Lisp form is the body of the effective method; it is augmented with an appropriate lambda-list to make it a function.

The role of each method in the effective method is determined by its method qualifiers and the specificity of the method. A qualifier serves to mark a method, and the meaning of a qualifier is determined by the way that these marks are used by this step of the procedure. If an applicable method has an unrecognized qualifier, this step signals an error and does not include that method in the effective method.

When standard method combination is used together with qualified methods, the effective method is produced as described in section 27.1.7.

Another type of method combination can be specified by using the `:method-combination` option of `defgeneric` or of any of the other forms that specify generic function options. In this way this step of the procedure can be customized.

New types of method combination can be defined by using the `define-method-combination` macro.

The meta-object level also offers a mechanism for defining new types of method combination. The generic function `compute-effective-method` receives as arguments the generic function, the method combination object, and the sorted list of applicable methods. It returns the Lisp form that defines the effective method. A method for `compute-effective-method` can be defined directly by using `defmethod` or indirectly by using `define-method-combination`. A *method combination object* is an object that encapsulates the method combination type and options specified by the `:method-combination` option to forms that specify generic function options.

Заметка для реализации: In the simplest implementation, the generic function would compute the effective method each time it was called. In practice, this will be too inefficient for some implementations. Instead, these implementations might employ a variety of optimizations of the three-step procedure. Some illustrative examples of such optimizations are the following:

- Use a hash table keyed by the class of the arguments to store the effective method.
- Compile the effective method and save the resulting compiled function in a table.
- Recognize the Lisp form as an instance of a pattern of control structure and substitute a closure that implements that structure.
- Examine the parameter specializers of all methods for the generic function and enumerate all possible effective methods. Combine the effective methods, together with code to select from among them, into a single function and compile that function. Call that function whenever the generic function is called.

Standard Method Combination

Standard method combination is supported by the class `standard-generic-function`. It is used if no other type of method combination is specified or if the built-in method combination type `standard` is specified.

Primary methods define the main action of the effective method, while *auxiliary methods* modify that action in one of three ways. A primary method has no method qualifiers.

An auxiliary method is a method whose method qualifier is `:before`, `:after`, or `:around`. Standard method combination allows no more than one qualifier per method; if a method definition specifies more than one qualifier per method, an error is signaled.

- A `:before` method has the keyword `:before` as its only qualifier. A `:before` method specifies code that is to be run before any primary method.
- An `:after` method has the keyword `:after` as its only qualifier. An `:after` method specifies code that is to be run after primary methods.
- An `:around` method has the keyword `:around` as its only qualifier. An `:around` method specifies code that is to be run instead of other applicable methods but that is able to cause some of them to be run.

The semantics of standard method combination are as follows:

- If there are any `:around` methods, the most specific `:around` method is called. It supplies the value or values of the generic function.
- Inside the body of an `:around` method, `call-next-method` can be used to call the next method. When the next method returns, the `:around` method can execute more code, perhaps based on the returned value or values. The generic function `no-next-method` is invoked if `call-next-method` is used and there is no applicable method to call. The function `next-method-p` may be used to determine whether a next method exists.

- If an `:around` method invokes `call-next-method`, the next most specific `:around` method is called, if one is applicable. If there are no `:around` methods or if `call-next-method` is called by the least specific `:around` method, the other methods are called as follows:

  - All the `:before` methods are called, in most-specific-first order. Their values are ignored. An error is signaled if `call-next-method` is used in a `:before` method.
  - The most specific primary method is called. Inside the body of a primary method, `call-next-method` may be used to call the next most specific primary method. When that method returns, the previous primary method can execute more code, perhaps based on the returned value or values. The generic function `no-next-method` is invoked if `call-next-method` is used and there are no more applicable primary methods. The function `next-method-p` may be used to determine whether a next method exists. If `call-next-method` is not used, only the most specific primary method is called.
  - All the `:after` methods are called in most-specific-last order. Their values are ignored. An error is signaled if `call-next-method` is used in an `:after` method.

- If no `:around` methods were invoked, the most specific primary method supplies the value or values returned by the generic function. The value or values returned by the invocation of `call-next-method` in the least specific `:around` method are those returned by the most specific primary method.

In standard method combination, if there is an applicable method but no applicable primary method, an error is signaled.

The `:before` methods are run in most-specific-first order and the `:after` methods are run in least-specific-first order. The design rationale for this difference can be illustrated with an example. Suppose class C_1 modifies the behavior of its superclass, C_2, by adding `:before` and `:after` methods. Whether the behavior of the class C_2 is defined directly by methods on C_2 or is inherited from its superclasses does not affect the relative order of invocation of methods on instances of the class C_1. Class C_1's `:before` method runs

before all of class C_2's methods. Class C_1's `:after` method runs after all of class C_2's methods.

By contrast, all `:around` methods run before any other methods run. Thus a less specific `:around` method runs before a more specific primary method.

If only primary methods are used and if `call-next-method` is not used, only the most specific method is invoked; that is, more specific methods shadow more general ones.

Declarative Method Combination

The macro `define-method-combination` defines new forms of method combination. It provides a mechanism for customizing the production of the effective method. The default procedure for producing an effective method is described in section 27.1.7. There are two forms of `define-method-combination`. The short form is a simple facility; the long form is more powerful and more verbose. The long form resembles `defmacro` in that the body is an expression that computes a Lisp form; it provides mechanisms for implementing arbitrary control structures within method combination and for arbitrary processing of method qualifiers. The syntax and use of both forms of `define-method-combination` are explained in section 27.2.

Built-in Method Combination Types

The Объектная система Common Lisp'a provides a set of built-in method combination types. To specify that a generic function is to use one of these method combination types, the name of the method combination type is given as the argument to the `:method-combination` option to `defgeneric` or to the `:method-combination` option to any of the other forms that specify generic function options.

The names of the built-in method combination types are `+`, `and`, `append`, `list`, `max`, `min`, `nconc`, `or`, `progn`, and `standard`.

The semantics of the `standard` built-in method combination type were described in section 27.1.7. The other built-in method combination types are called *simple built-in method combination types.*

The simple built-in method combination types act as though they were defined by the short form of `define-method-combination`. They recognize two roles for methods:

- An `:around` method has the keyword symbol `:around` as its sole qualifier. The meaning of `:around` methods is the same as in standard method combination. Use of the functions `call-next-method` and `next-method-p` is supported in `:around` methods.
- A primary method has the name of the method combination type as its sole qualifier. For example, the built-in method combination type `and` recognizes methods whose sole qualifier is `and`; these are primary methods. Use of the functions `call-next-method` and `next-method-p` is not supported in primary methods.

The semantics of the simple built-in method combination types are as follows:

- If there are any `:around` methods, the most specific `:around` method is called. It supplies the value or values of the generic function.
- Inside the body of an `:around` method, the function `call-next-method` can be used to call the next method. The generic function `no-next-method` is invoked if `call-next-method` is used and there is no applicable method to call. The function `next-method-p` may be used to determine whether a next method exists. When the next method returns, the `:around` method can execute more code, perhaps based on the returned value or values.
- If an `:around` method invokes `call-next-method`, the next most specific `:around` method is called, if one is applicable. If there are no `:around` methods or if `call-next-method` is called by the least specific `:around` method, a Lisp form derived from the name of the built-in method combination type and from the list of applicable primary methods is evaluated to produce the value of the generic function. Suppose the name of the method combination type is *operator* and the call to the generic function is of the form

  (*generic-function* a_1 ... a_n)

Let $M_1, \ldots, M_k$ be the applicable primary methods in order; then the derived Lisp form is

(*operator* $\langle M_1\ a_1 \ldots a_n \rangle$ ... $\langle M_k\ a_1 \ldots a_n \rangle$)

If the expression $\langle M_i\ a_1 \ldots a_n \rangle$ is evaluated, the method M_i will be applied to the arguments $a_1 \ldots a_n$. For example, if *operator* is `or`, the expression $\langle M_i\ a_1 \ldots a_n \rangle$ is evaluated only if $\langle M_j\ a_1 \ldots a_n \rangle$, $1 \leq j < i$, returned `nil`.

The default order for the primary methods is `:most-specific-first`. However, the order can be reversed by supplying `:most-specific-last` as the second argument to the `:method-combination` option.

The simple built-in method combination types require exactly one qualifier per method. An error is signaled if there are applicable methods with no qualifiers or with qualifiers that are not supported by the method combination type. An error is signaled if there are applicable `:around` methods and no applicable primary methods.

27.1.8 Meta-objects

The implementation of the Объектная система manipulates classes, methods, and generic functions. The meta-object protocol specifies a set of generic functions defined by methods on classes; the behavior of those generic functions defines the behavior of the Объектная система. The instances of the classes on which those methods are defined are called *meta-objects*. Programming at the meta-object protocol level involves defining new classes of meta-objects along with methods specialized on these classes.

Metaclasses

The *metaclass* of an object is the class of its class. The metaclass determines the representation of instances of its instances and the forms of inheritance used by its instances for slot descriptions and method inheritance. The metaclass mechanism can be used to provide particular forms of optimization or to tailor the Объектная система Common Lisp'a for particular uses. The protocol for defining metaclasses is discussed in the third part of the CLOS specification, The Объектная система Common Lisp'a Meta-Object Protocol. [The third part has not yet been approved by X3J13 for inclusion in the forthcoming Common Lisp standard and is not included in this book.—GLS]

Standard Metaclasses

The Объектная система Common Lisp'a provides a number of predefined metaclasses. These include the classes `standard-class`, `built-in-class`, and `structure-class`:

- The class `standard-class` is the default class of classes defined by `defclass`.
- The class `built-in-class` is the class whose instances are classes that have special implementations with restricted capabilities. Any class that corresponds to a standard Common Lisp type might be an instance of `built-in-class`. The predefined Common Lisp type specifiers that are required to have corresponding classes are listed in table 27.1. It is implementation-dependent whether each of these classes is implemented as a built-in class.
- All classes defined by means of `defstruct` are instances of `structure-class`.

Standard Meta-objects

The Объектная система supplies a standard set of meta-objects, called *standard meta-objects*. These include the class `standard-object` and instances of the classes `standard-method`, `standard-generic-function`, and `method-combination`.

- The class `standard-method` is the default class of methods that are defined by the forms `defmethod`, `defgeneric`, `generic-function`, `generic-flet`, `generic-labels`, and `with-added-methods`.
- The class `standard-generic-function` is the default class of generic functions defined by the forms `defmethod`, `defgeneric`, `generic-function`, `generic-flet`, `generic-labels`, `with-added-methods`, and `defclass`.
- The class named `standard-object` is an instance of the class `standard-class` and is a superclass of every class that is an instance of `standard-class` except itself.
- Every method combination object is an instance of a subclass of the class `method-combination`.

27.1.9 Object Creation and Initialization

The generic function `make-instance` creates and returns a new instance of a class. The first argument is a class or the name of a class, and the remaining arguments form an *initialization argument* list.

The initialization of a new instance consists of several distinct steps, including the following: combining the explicitly supplied initialization arguments with default values for the unsupplied initialization arguments, checking the validity of the initialization arguments, allocating storage for the instance, filling slots with values, and executing user-supplied methods that perform additional initialization. Each step of `make-instance` is implemented by a generic function to provide a mechanism for customizing that step. In addition, `make-instance` is itself a generic function and thus also can be customized.

The Объектная система specifies system-supplied primary methods for each step and thus specifies a well-defined standard behavior for the entire initialization process. The standard behavior provides four simple mechanisms for controlling initialization:

- Declaring a symbol to be an initialization argument for a slot. An initialization argument is declared by using the `:initarg` slot option to `defclass`. This provides a mechanism for supplying a value for a slot in a call to `make-instance`.

- Supplying a default value form for an initialization argument. Default value forms for initialization arguments are defined by using the `:default-initargs` class option to `defclass`. If an initialization argument is not explicitly provided as an argument to `make-instance`, the default value form is evaluated in the lexical environment of the `defclass` form that defined it, and the resulting value is used as the value of the initialization argument.

- Supplying a default initial value form for a slot. A default initial value form for a slot is defined by using the `:initform` slot option to `defclass`. If no initialization argument associated with that slot is given as an argument to `make-instance` or is defaulted by `:default-initargs`, this default initial value form is evaluated in the lexical environment of the `defclass` form that defined it, and the resulting value is stored in the slot. The `:initform` form for a local slot may be used when creating an instance, when updating an instance to conform to a redefined class, or when updating an instance to conform to the definition of a different class. The `:initform` form for a shared slot may be used when defining or re-defining the class.

- Defining methods for `initialize-instance` and `shared-initialize`. The slot-filling behavior described above is implemented by a system-supplied primary method for `initialize-instance` which invokes `shared-initialize`. The generic function `shared-initialize` implements the parts of initialization shared by these four situations: when making an instance, when re-initializing an instance, when updating an instance to conform to a redefined class, and when updating an

instance to conform to the definition of a different class. The system-supplied primary method for `shared-initialize` directly implements the slot-filling behavior described above, and `initialize-instance` simply invokes `shared-initialize`.

Initialization Arguments

An initialization argument controls object creation and initialization. It is often convenient to use keyword symbols to name initialization arguments, but the name of an initialization argument can be any symbol, including `nil`. An initialization argument can be used in two ways: to fill a slot with a value or to provide an argument for an initialization method. A single initialization argument can be used for both purposes.

An *initialization argument list* is a list of alternating initialization argument names and values. Its structure is identical to a property list and also to the portion of an argument list processed for `&key` parameters. As in those lists, if an initialization argument name appears more than once in an initialization argument list, the leftmost occurrence supplies the value and the remaining occurrences are ignored. The arguments to `make-instance` (after the first argument) form an initialization argument list. Error checking of initialization argument names is disabled if the keyword argument pair whose keyword is `:allow-other-keys` and whose value is non-`nil` appears in the initialization argument list.

An initialization argument can be associated with a slot. If the initialization argument has a value in the initialization argument list, the value is stored into the slot of the newly created object, overriding any `:initform` form associated with the slot. A single initialization argument can initialize more than one slot. An initialization argument that initializes a shared slot stores its value into the shared slot, replacing any previous value.

An initialization argument can be associated with a method. When an object is created and a particular initialization argument is supplied, the generic functions `initialize-instance`, `shared-initialize`, and `allocate-instance` are called with that initialization argument's name and value as a keyword argument pair. If a value for the initialization argument is not supplied in the initialization argument list, the method's lambda-list supplies a default value.

Initialization arguments are used in four situations: when making an instance, when re-initializing an instance, when updating an instance to conform to a redefined class, and when updating an instance to conform to the definition of a different class.

Because initialization arguments are used to control the creation and initialization of an instance of some particular class, we say that an initialization argument is "an initialization argument for" that class.

Declaring the Validity of Initialization Arguments

Initialization arguments are checked for validity in each of the four situations that use them. An initialization argument may be valid in one situation and not another. For example, the system-supplied primary method for `make-instance` defined for the class `standard-class` checks the validity of its initialization arguments and signals an error if an initialization argument is supplied that is not declared valid in that situation.

There are two means of declaring initialization arguments valid.

- Initialization arguments that fill slots are declared valid by the `:initarg` slot option to `defclass`. The `:initarg` slot option is inherited from superclasses. Thus the set of valid initialization arguments that fill slots for a class is the union of the initialization arguments that fill slots declared valid by that class and its superclasses. Initialization arguments that fill slots are valid in all four contexts.
- Initialization arguments that supply arguments to methods are declared valid by defining those methods. The keyword name of each keyword parameter specified in the method's lambda-list becomes an initialization argument for all classes for which the method is applicable. Thus method inheritance controls the set of valid initialization arguments that supply arguments to methods. The generic functions for which method definitions serve to declare initialization arguments valid are as follows:
  - Making an instance of a class: `allocate-instance`, `initialize-instance`, and `shared-initialize`. Initializa-

tion arguments declared valid by these methods are valid when making an instance of a class.

- Re-initializing an instance: the functions `reinitialize-instance` and `shared-initialize`. Initialization arguments declared valid by these methods are valid when re-initializing an instance.
- Updating an instance to conform to a redefined class: `update-instance-for-redefined-class` and `shared-initialize`. Initialization arguments declared valid by these methods are valid when updating an instance to conform to a redefined class.
- Updating an instance to conform to the definition of a different class: `update-instance-for-different-class` and `shared-initialize`. Initialization arguments declared valid by these methods are valid when updating an instance to conform to the definition of a different class.

The set of valid initialization arguments for a class is the set of valid initialization arguments that either fill slots or supply arguments to methods, along with the predefined initialization argument `:allow-other-keys`. The default value for `:allow-other-keys` is `nil`. The meaning of `:allow-other-keys` is the same here as when it is passed to an ordinary function.

Defaulting of Initialization Arguments

A *default value form* can be supplied for an initialization argument by using the `:default-initargs` class option. If an initialization argument is declared valid by some particular class, its default value form might be specified by a different class. In this case `:default-initargs` is used to supply a default value for an inherited initialization argument.

The `:default-initargs` option is used only to provide default values for initialization arguments; it does not declare a symbol as a valid initialization argument name. Furthermore, the `:default-initargs` option is used only to provide default values for initialization arguments when making an instance.

The argument to the `:default-initargs` class option is a list of alternating initialization argument names and forms. Each form is the default

value form for the corresponding initialization argument. The default value form of an initialization argument is used and evaluated only if that initialization argument does not appear in the arguments to `make-instance` and is not defaulted by a more specific class. The default value form is evaluated in the lexical environment of the `defclass` form that supplied it; the result is used as the initialization argument's value.

The initialization arguments supplied to `make-instance` are combined with defaulted initialization arguments to produce a *defaulted initialization argument list*. A defaulted initialization argument list is a list of alternating initialization argument names and values in which unsupplied initialization arguments are defaulted and in which the explicitly supplied initialization arguments appear earlier in the list than the defaulted initialization arguments. Defaulted initialization arguments are ordered according to the order in the class precedence list of the classes that supplied the default values.

There is a distinction between the purposes of the `:default-initargs` and the `:initform` options with respect to the initialization of slots. The `:default-initargs` class option provides a mechanism for the user to give a default value form for an initialization argument without knowing whether the initialization argument initializes a slot or is passed to a method. If that initialization argument is not explicitly supplied in a call to `make-instance`, the default value form is used, just as if it had been supplied in the call. In contrast, the `:initform` slot option provides a mechanism for the user to give a default initial value form for a slot. An `:initform` form is used to initialize a slot only if no initialization argument associated with that slot is given as an argument to `make-instance` or is defaulted by `:default-initargs`.

The order of evaluation of default value forms for initialization arguments and the order of evaluation of `:initform` forms are undefined. If the order of evaluation matters, use `initialize-instance` or `shared-initialize` methods.

Rules for Initialization Arguments

The `:initarg` slot option may be specified more than once for a given slot. The following rules specify when initialization arguments may be multiply defined:

- A given initialization argument can be used to initialize more than one

slot if the same initialization argument name appears in more than one `:initarg` slot option.

- A given initialization argument name can appear in the lambda-list of more than one initialization method.
- A given initialization argument name can appear both in an `:initarg` slot option and in the lambda-list of an initialization method.

If two or more initialization arguments that initialize the same slot are given in the arguments to `make-instance`, the leftmost of these initialization arguments in the initialization argument list supplies the value, even if the initialization arguments have different names.

If two or more different initialization arguments that initialize the same slot have default values and none is given explicitly in the arguments to `make-instance`, the initialization argument that appears in a `:default-initargs` class option in the most specific of the classes supplies the value. If a single `:default-initargs` class option specifies two or more initialization arguments that initialize the same slot and none is given explicitly in the arguments to `make-instance`, the leftmost argument in the `:default-initargs` class option supplies the value, and the values of the remaining default value forms are ignored.

Initialization arguments given explicitly in the arguments to `make-instance` appear to the left of defaulted initialization arguments. Suppose that the classes C_1 and C_2 supply the values of defaulted initialization arguments for different slots, and suppose that C_1 is more specific than C_2; then the defaulted initialization argument whose value is supplied by C_1 is to the left of the defaulted initialization argument whose value is supplied by C_2 in the defaulted initialization argument list. If a single `:default-initargs` class option supplies the values of initialization arguments for two different slots, the initialization argument whose value is specified farther to the left in the `default-initargs` class option appears farther to the left in the defaulted initialization argument list.

If a slot has both an `:initform` form and an `:initarg` slot option, and the initialization argument is defaulted using `:default-initargs` or is supplied to `make-instance`, the captured `:initform` form is neither used nor evaluated.

The following is an example of the preceding rules:

```
(defclass q () ((x :initarg a)))

(defclass r (q) ((x :initarg b))
  (:default-initargs a 1 b 2))
```

Form	Defaulted Initialization Argument List	Contents of Slot
`(make-instance 'r)`	`(a 1 b 2)`	`1`
`(make-instance 'r 'a 3)`	`(a 3 b 2)`	`3`
`(make-instance 'r 'b 4)`	`(b 4 a 1)`	`4`
`(make-instance 'r 'a 1 'a 2)`	`(a 1 a 2 b 2)`	`1`

Shared-Initialize

The generic function `shared-initialize` is used to fill the slots of an instance using initialization arguments and `:initform` forms when an instance is created, when an instance is re-initialized, when an instance is updated to conform to a redefined class, and when an instance is updated to conform to a different class. It uses standard method combination. It takes the following arguments: the instance to be initialized, a specification of a set of names of slots accessible in that instance, and any number of initialization arguments. The arguments after the first two must form an initialization argument list.

The second argument to `shared-initialize` may be one of the following:

- It can be a list of slot names, which specifies the set of those slot names.
- It can be `nil`, which specifies the empty set of slot names.
- It can be the symbol `t`, which specifies the set of all of the slots.

There is a system-supplied primary method for `shared-initialize` whose first parameter specializer is the class `standard-object`. This method behaves as follows on each slot, whether shared or local:

- If an initialization argument in the initialization argument list specifies a value for that slot, that value is stored into the slot, even if a value has already been stored in the slot before the method is run. The affected slots are independent of which slots are indicated by the second argument to `shared-initialize`.

- Any slots indicated by the second argument that are still unbound at this point are initialized according to their `:initform` forms. For any such slot that has an `:initform` form, that form is evaluated in the lexical environment of its defining `defclass` form and the result is stored into the slot. For example, if a `:before` method stores a value in the slot, the `:initform` form will not be used to supply a value for the slot. If the second argument specifies a name that does not correspond to any slots accessible in the instance, the results are unspecified.

- The rules mentioned in section 27.1.9 are obeyed.

The generic function `shared-initialize` is called by the system-supplied primary methods for the generic functions `initialize-instance`, `reinitialize-instance`, `update-instance-for-different-class`, and `update-instance-for-redefined-class`. Thus methods can be written for `shared-initialize` to specify actions that should be taken in all of these contexts.

Initialize-Instance

The generic function `initialize-instance` is called by `make-instance` to initialize a newly created instance. It uses standard method combination. Methods for `initialize-instance` can be defined in order to perform any initialization that cannot be achieved with the simple slot-filling mechanisms.

During initialization, `initialize-instance` is invoked after the following actions have been taken:

- The defaulted initialization argument list has been computed by combining the supplied initialization argument list with any default initialization arguments for the class.
- The validity of the defaulted initialization argument list has been checked. If any of the initialization arguments has not been declared valid, an error is signaled.
- A new instance whose slots are unbound has been created.

The generic function `initialize-instance` is called with the new instance and the defaulted initialization arguments. There is a system-supplied primary method for `initialize-instance` whose parameter specializer is the class `standard-object`. This method calls the generic function `shared-initialize` to fill in the slots according to the initialization arguments and the `:initform` forms for the slots; the generic function `shared-initialize` is called with the following arguments: the instance, `t`, and the defaulted initialization arguments.

Note that `initialize-instance` provides the defaulted initialization argument list in its call to `shared-initialize`, so the first step performed by the system-supplied primary method for `shared-initialize` takes into account both the initialization arguments provided in the call to `make-instance` and the defaulted initialization argument list.

Methods for `initialize-instance` can be defined to specify actions to be taken when an instance is initialized. If only `:after` methods for `initialize-instance` are defined, they will be run after the system-supplied primary method for initialization and therefore they will not interfere with the default behavior of `initialize-instance`.

The Объектная система provides two functions that are useful in the bodies of `initialize-instance` methods. The function `slot-boundp` returns a boolean value that indicates whether a specified slot has a value; this provides a mechanism for writing `:after` methods for `initialize-instance` that initialize slots only if they have not already been initialized. The function `slot-makunbound` causes the slot to have no value.

Definitions of Make-Instance and Initialize-Instance

The generic function `make-instance` behaves as if it were defined as follows, except that certain optimizations are permitted:

```
(defmethod make-instance ((class standard-class) &rest initargs)
  (setq initargs (default-initargs class initargs))
  ...
  (let ((instance (apply #'allocate-instance class initargs)))
    (apply #'initialize-instance instance initargs)
    instance))

(defmethod make-instance ((class-name symbol) &rest initargs)
  (apply #'make-instance (find-class class-name) initargs))
```

The elided code in the definition of `make-instance` checks the supplied initialization arguments to determine whether an initialization argument was supplied that neither filled a slot nor supplied an argument to an applicable method. This check could be implemented using the generic functions `class-prototype`, `compute-applicable-methods`, `function-keywords`, and `class-slot-initargs`. See the third part of the Объектная система Common Lisp'a specification for a description of this initialization argument check. [The third part has not yet been approved by X3J13 for inclusion in the forthcoming Common Lisp standard and is not included in this book.—GLS]

The generic function `initialize-instance` behaves as if it were defined as follows, except that certain optimizations are permitted:

```
(defmethod initialize-instance
           ((instance standard-object) &rest initargs)
  (apply #'shared-initialize instance t initargs)))
```

These procedures can be customized at either the Programmer Interface level, the meta-object level, or both.

Customizing at the Programmer Interface level includes using the `:initform`, `:initarg`, and `:default-initargs` options to `defclass`, as well as defining methods for `make-instance` and `initialize-instance`. It is also possible to define methods for `shared-initialize`, which would be invoked by the generic functions

`reinitialize-instance`, `update-instance-for-redefined-class`, `update-instance-for-different-class`, and `initialize-instance`. The meta-object level supports additional customization by allowing methods to be defined on `make-instance`, `default-initargs`, and `allocate-instance`. Parts 2 and 3 of the Объектная система Common Lisp'a specification document each of these generic functions and the system-supplied primary methods. [The third part has not yet been approved by X3J13 for inclusion in the forthcoming Common Lisp standard and is not included in this book.—GLS]

Implementations are permitted to make certain optimizations to `initialize-instance` and `shared-initialize`. The description of `shared-initialize` in section 27.2 mentions the possible optimizations.

Because of optimization, the check for valid initialization arguments might not be implemented using the generic functions `class-prototype`, `compute-applicable-methods`, `function-keywords`, and `class-slot-initargs`. In addition, methods for the generic function `default-initargs` and the system-supplied primary methods for `allocate-instance`, `initialize-instance`, and `shared-initialize` might not be called on every call to `make-instance` or might not receive exactly the arguments that would be expected.

27.1.10 Redefining Classes

A class that is an instance of `standard-class` can be redefined if the new class will also be an instance of `standard-class`. Redefining a class modifies the existing class object to reflect the new class definition; it does not create a new class object for the class. Any method object created by a `:reader`, `:writer`, or `:accessor` option specified by the old `defclass` form is removed from the corresponding generic function. Methods specified by the new `defclass` form are added.

When the class C is redefined, changes are propagated to its instances and to instances of any of its subclasses. Updating such an instance occurs at an implementation-dependent time, but no later than the next time a slot of that instance is read or written. Updating an instance does not change its identity as defined by the `eq` function. The updating process may change the slots of that particular instance, but it does not create a new instance. Whether updating an instance consumes storage is implementation-dependent.

Note that redefining a class may cause slots to be added or deleted. If a class is redefined in a way that changes the set of local slots accessible in instances, the instances will be updated. It is implementation-dependent whether instances are updated if a class is redefined in a way that does not change the set of local slots accessible in instances.

The value of a slot that is specified as shared both in the old class and in the new class is retained. If such a shared slot was unbound in the old class, it will be unbound in the new class. Slots that were local in the old class and that are shared in the new class are initialized. Newly added shared slots are initialized.

Each newly added shared slot is set to the result of evaluating the captured `:initform` form for the slot that was specified in the `defclass` form for the new class. If there is no `:initform` form, the slot is unbound.

If a class is redefined in such a way that the set of local slots accessible in an instance of the class is changed, a two-step process of updating the instances of the class takes place. The process may be explicitly started by invoking the generic function `make-instances-obsolete`. This two-step process can happen in other circumstances in some implementations. For example, in some implementations this two-step process will be triggered if the order of slots in storage is changed.

The first step modifies the structure of the instance by adding new local slots and discarding local slots that are not defined in the new version of the class. The second step initializes the newly added local slots and performs any other user-defined actions. These steps are further specified in the next two sections.

Modifying the Structure of Instances

The first step modifies the structure of instances of the redefined class to conform to its new class definition. Local slots specified by the new class definition that are not specified as either local or shared by the old class are added, and slots not specified as either local or shared by the new class definition that are specified as local by the old class are discarded. The names of these added and discarded slots are passed as arguments to `update-instance-for-redefined-class` as described in the next section.

The values of local slots specified by both the new and old classes are retained. If such a local slot was unbound, it remains unbound.

The value of a slot that is specified as shared in the old class and as local in the new class is retained. If such a shared slot was unbound, the local slot will be unbound.

Initializing Newly Added Local Slots

The second step initializes the newly added local slots and performs any other user-defined actions. This step is implemented by the generic function `update-instance-for-redefined-class`, which is called after completion of the first step of modifying the structure of the instance.

The generic function `update-instance-for-redefined-class` takes four required arguments: the instance being updated after it has undergone the first step, a list of the names of local slots that were added, a list of the names of local slots that were discarded, and a property list containing the slot names and values of slots that were discarded and had values. Included among the discarded slots are slots that were local in the old class and that are shared in the new class.

The generic function `update-instance-for-redefined-class` also takes any number of initialization arguments. When it is called by the system to update an instance whose class has been redefined, no initialization arguments are provided.

There is a system-supplied primary method for the generic function `update-instance-for-redefined-class` whose parameter specializer for its instance argument is the class `standard-object`. First this method checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared valid (see section 27.1.9.) Then it calls the generic function `shared-initialize` with the following arguments: the instance, the list of names of the newly added slots, and the initialization arguments it received.

Customizing Class Redefinition

Methods for `update-instance-for-redefined-class` may be defined to specify actions to be taken when an instance is updated. If only `:after` methods for `update-instance-for-redefined-class` are defined, they will be run after the system-supplied primary method for initial-

ization and therefore will not interfere with the default behavior of `update-instance-for-redefined-class`. Because no initialization arguments are passed to `update-instance-for-redefined-class` when it is called by the system, the `:initform` forms for slots that are filled by `:before` methods for `update-instance-for-redefined-class` will not be evaluated by `shared-initialize`.

Methods for `shared-initialize` may be defined to customize class redefinition (see section 27.1.9).

Extensions

There are two allowed extensions to class redefinition:

- The Объектная система may be extended to permit the new class to be an instance of a metaclass other than the metaclass of the old class.
- The Объектная система may be extended to support an updating process when either the old or the new class is an instance of a class other than `standard-class` that is not a built-in class.

27.1.11 Changing the Class of an Instance

The function `change-class` can be used to change the class of an instance from its current class, C_{from}, to a different class, C_{to}; it changes the structure of the instance to conform to the definition of the class C_{to}.

Note that changing the class of an instance may cause slots to be added or deleted.

When `change-class` is invoked on an instance, a two-step updating process takes place. The first step modifies the structure of the instance by adding new local slots and discarding local slots that are not specified in the new version of the instance. The second step initializes the newly added local slots and performs any other user-defined actions. These steps are further described in the following two sections.

Modifying the Structure of an Instance

In order to make an instance conform to the class C_{to}, local slots specified by the class C_{to} that are not specified by the class C_{from} are added, and local slots not specified by the class C_{to} that are specified by the class C_{from} are discarded.

The values of local slots specified by both the class C_{to} and the class C_{from} are retained. If such a local slot was unbound, it remains unbound.

The values of slots specified as shared in the class C_{from} and as local in the class C_{to} are retained.

This first step of the update does not affect the values of any shared slots.

Initializing Newly Added Local Slots

The second step of the update initializes the newly added slots and performs any other user-defined actions. This step is implemented by the generic function `update-instance-for-different-class`. The generic function `update-instance-for-different-class` is invoked by `change-class` after the first step of the update has been completed.

The generic function `update-instance-for-different-class` is invoked on two arguments computed by `change-class`. The first argument passed is a copy of the instance being updated and is an instance of the class C_{from}; this copy has dynamic extent within the generic function `change-class`. The second argument is the instance as updated so far by `change-class` and is an instance of the class C_{to}.

The generic function `update-instance-for-different-class` also takes any number of initialization arguments. When it is called by `change-class`, no initialization arguments are provided.

There is a system-supplied primary method for the generic function `update-instance-for-different-class` that has two parameter specializers, each of which is the class `standard-object`. First this method checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared valid (see section 27.1.9). Then it calls the generic function `shared-initialize` with the following arguments: the instance, a list of names of the newly added slots, and the initialization arguments it received.

Customizing the Change of Class of an Instance

Methods for `update-instance-for-different-class` may be defined to specify actions to be taken when an instance is updated. If only `:after` methods for `update-instance-for-different-class` are defined, they will be run after the system-supplied primary method for initialization and will not interfere with the default behavior of `update-instance-for-different-class`. Because no initialization arguments are passed to `update-instance-for-different-class` when it is called by `change-class`, the `:initform` forms for slots that are filled by `:before` methods for `update-instance-for-different-class` will not be evaluated by `shared-initialize`.

Methods for `shared-initialize` may be defined to customize class redefinition (see section 27.1.9).

27.1.12 Reinitializing an Instance

The generic function `reinitialize-instance` may be used to change the values of slots according to initialization arguments.

The process of reinitialization changes the values of some slots and performs any user-defined actions.

Reinitialization does not modify the structure of an instance to add or delete slots, and it does not use any `:initform` forms to initialize slots.

The generic function `reinitialize-instance` may be called directly. It takes one required argument, the instance. It also takes any number of initialization arguments to be used by methods for `reinitialize-instance` or for `shared-initialize`. The arguments after the required instance must form an initialization argument list.

There is a system-supplied primary method for `reinitialize-instance` whose parameter specializer is the class `standard-object`. First this method checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared valid (see section 27.1.9). Then it calls the generic function `shared-initialize` with the following arguments: the instance, `nil`, and the initialization arguments it received.

Customizing Reinitialization

Methods for the generic function `reinitialize-instance` may be defined to specify actions to be taken when an instance is updated. If only `:after` methods for `reinitialize-instance` are defined, they will be run after the system-supplied primary method for initialization and therefore will not interfere with the default behavior of `reinitialize-instance`.

Methods for `shared-initialize` may be defined to customize class redefinition (see section 27.1.9).

27.2 Functions in the Programmer Interface

This section describes the functions, macros, special operators, and generic functions provided by the Объектная система Common Lisp'a Programmer Interface. The Programmer Interface comprises the functions and macros that are sufficient for writing most object-oriented programs.

This section is reference material that requires an understanding of the basic concepts of the Common Lisp Object System. The functions are arranged in alphabetical order for convenient reference.

The description of each function, macro, special operator, and generic function includes its purpose, its syntax, the semantics of its arguments and returned values, and often an example and cross-references to related functions.

The syntax description for a function, macro, or special operator describes its parameters. The description of a generic function includes descriptions of the methods that are defined on that generic function by the Объектная система Common Lisp'a. A *method signature* is used to describe the parameters and parameter specializers for each method.

The following is an example of the format for the syntax description of a generic function with the method signature for one primary method:

[Generic function] **f** *x y* **&optional** *z* **&key** *:k*
[Primary method] **f** (*x* class) (*y* t) **&optional** *z* **&key** *:k*

This description indicates that the generic function `f` has two required

parameters, x and y. In addition, there is an optional parameter z and a keyword parameter `:k`.

The method signature indicates that this method on the generic function `f` has two required parameters, x, which must be an instance of the class `class`, and y, which can be any object. In addition, there is an optional parameter z and a keyword parameter `:k`. The signature also indicates that this method on `f` is a primary method and has no qualifiers.

The syntax description for a generic function describes the lambda-list of the generic function itself, while the method signatures describe the lambda-lists of the defined methods.

The generic functions described in this book are all standard generic functions. They all use standard method combination.

Any implementation of the Объектная система Common Lisp'a is allowed to provide additional methods on the generic functions described here.

It is useful to categorize the functions and macros according to their role in this standard:

- *Tools used for simple object-oriented programming*

  These tools allow for defining new classes, methods, and generic functions and for making instances. Some tools used within method bodies are also listed here. Some of the macros listed here have a corresponding function that performs the same task at a lower level of abstraction.

```
call-next-method  initialize-instance
change-class      make-instance
defclass          next-method-p
defgeneric        slot-boundp
defmethod         slot-value
generic-flet      with-accessors
generic-function  with-added-methods
generic-labels    with-slots
```

- *Functions underlying the commonly used macros*

```
add-method                    reinitialize-instance
class-name                    remove-method
compute-applicable-methods    shared-initialize
ensure-generic-function       slot-exists-p
find-class                    slot-makunbound
find-method                   slot-missing
function-keywords             slot-unbound
make-instances-obsolete       update-instance-for-different-class
no-applicable-method          update-instance-for-redefined-class
no-next-method
```

- *Tools for declarative method combination*

```
call-method                 method-combination-error
define-method-combination   method-qualifiers
invalid-method-error
```

- *General Common Lisp support tools*

```
class-of       print-object
documentation  symbol-macrolet
```

[Note that `describe` appeared in this list in the original CLOS proposal [5, 7], but X3J13 voted in March 1989 not to make `describe` a generic function after all (see `describe-object`).—GLS]

[At this point the original CLOS report contained a description of the [[ ]] and ↓ notation; that description is omitted here. I have adopted the notation for use throughout this book. It is described in section 1.2.5.—GLS]

[Generic function] **add-method** *generic-function method*
[Primary method] **add-method**
(*generic-function* standard-generic-function) (*method* method)

The generic function `add-method` adds a method to a generic function. It destructively modifies the generic function and returns the modified generic function as its result.

The *generic-function* argument is a generic function object.

The *method* argument is a method object. The lambda-list of the method function must be congruent with the lambda-list of the generic function, or an error is signaled.

The modified generic function is returned. The result of `add-method` is `eq` to the *generic-function* argument.

If the given method agrees with an existing method of the generic function on parameter specializers and qualifiers, the existing method is replaced. See section 27.1.6 for a definition of agreement in this context.

If the method object is a method object of another generic function, an error is signaled.

See section 27.1.6 as well as `defmethod`, `defgeneric`, `find-method`, and `remove-method`.

[Макрос] **call-method** method next-method-list

The macro `call-method` is used in method combination. This macro hides the implementation-dependent details of how methods are called. It can be used only within an effective method form, for the name `call-method` is defined only within the lexical scope of such a form.

The macro `call-method` invokes the specified method, supplying it with arguments and with definitions for `call-next-method` and for `next-method-p`. The arguments are the arguments that were supplied to the effective method form containing the invocation of `call-method`. The definitions of `call-next-method` and `next-method-p` rely on the list of method objects given as the second argument to `call-method`.

The `call-next-method` function available to the method that is the first subform will call the first method in the list that is the second subform. The `call-next-method` function available in that method, in turn, will call the second method in the list that is the second subform, and so on, until the list of next methods is exhausted.

The *method* argument is a method object; the *next-method-list* argument is a list of method objects.

A list whose first element is the symbol `make-method` and whose second element is a Lisp form can be used instead of a method object as the first subform of `call-method` or as an element of the second subform of `call-method`. Such a list specifies a method object whose method function has a body that is the given form.

The result of `call-method` is the value or values returned by the method invocation.

See `call-next-method`, `define-method-combination`, and `next-method-p`.

[Function] **call-next-method &rest** *args*

The function `call-next-method` can be used within the body of a method defined by a method-defining form to call the next method.

The function `call-next-method` returns the value or values returned by the method it calls. If there is no next method, the generic function `no-next-method` is called.

The type of method combination used determines which methods can invoke `call-next-method`. The standard method combination type allows `call-next-method` to be used within primary methods and `:around` methods.

The standard method combination type defines the next method according to the following rules:

- If `call-next-method` is used in an `:around` method, the next method is the next most specific `:around` method, if one is applicable.
- If there are no `:around` methods at all or if `call-next-method` is called by the least specific `:around` method, other methods are called as follows:
  - All the `:before` methods are called, in most-specific-first order. The function `call-next-method` cannot be used in `:before` methods.
  - The most specific primary method is called. Inside the body of a primary method, `call-next-method` may be used to pass control to the next most specific primary method. The generic function `no-next-method` is called if `call-next-method` is used and there are no more primary methods.
  - All the `:after` methods are called in most-specific-last order. The function `call-next-method` cannot be used in `:after` methods.

For further discussion of the use of `call-next-method`, see sections 27.1.7 and 27.1.7.

When `call-next-method` is called with no arguments, it passes the current method's original arguments to the next method. Neither argument defaulting, nor using `setq`, nor rebinding variables with the same names as parameters of the method affects the values `call-next-method` passes to the method it calls.

When `call-next-method` is called with arguments, the next method is called with those arguments. When providing arguments to `call-next-method`, the following rule must be satisfied or an error is signaled: The ordered set of methods applicable for a changed set of arguments for `call-next-method` must be the same as the ordered set of applicable methods for the original arguments to the generic function. Optimizations of the error checking are possible, but they must not change the semantics of `call-next-method`.

If `call-next-method` is called with arguments but omits optional arguments, the next method called defaults those arguments.

The function `call-next-method` returns the value or values returned by the method it calls.

Further computation is possible after `call-next-method` returns.

The definition of the function `call-next-method` has lexical scope (for it is defined only within the body of a method defined by a method-defining form) and indefinite extent.

For generic functions using a type of method combination defined by the short form of `define-method-combination`, `call-next-method` can be used in `:around` methods only.

The function `next-method-p` can be used to test whether or not there is a next method.

If `call-next-method` is used in methods that do not support it, an error is signaled.

See sections 27.1.7, 27.1.7, and 27.1.7 as well as the functions `define-method-combination`, `next-method-p`, and `no-next-method`.

[Generic function] **change-class** *instance new-class*
[Primary method] **change-class** (*instance* standard-object) (*new-class* standard-class)
[Primary method] **change-class** (*instance* t) (*new-class* symbol)

The generic function `change-class` changes the class of an instance to a new class. It destructively modifies and returns the instance.

If in the old class there is any slot of the same name as a local slot in the new class, the value of that slot is retained. This means that if the slot has a value, the value returned by `slot-value` after `change-class` is invoked is `eql` to the value returned by `slot-value` before `change-class` is invoked. Similarly, if the slot was unbound, it remains unbound. The other slots are initialized as described in section 27.1.11.

The *instance* argument is a Lisp object.

The *new-class* argument is a class object or a symbol that names a class.

If the second of the preceding methods is selected, that method invokes `change-class` on *instance* and `(find-class` *`new-class`*`)`.

The modified instance is returned. The result of `change-class` is `eq` to the *instance* argument.

Examples:

```
(defclass position () ())

(defclass x-y-position (position)
  ((x :initform 0 :initarg :x)
   (y :initform 0 :initarg :y)))

(defclass rho-theta-position (position)
  ((rho :initform 0)
   (theta :initform 0)))

(defmethod update-instance-for-different-class :before
        ((old x-y-position)
         (new rho-theta-position)
         &key)
```

```
  ;; Copy the position information from old to new to make new
  ;; be a rho-theta-position at the same position as old.
  (let ((x (slot-value old 'x))
        (y (slot-value old 'y)))
    (setf (slot-value new 'rho) (sqrt (+ (* x x) (* y y)))
          (slot-value new 'theta) (atan y x))))
```

```
;;; At this point an instance of the class x-y-position can be
;;; changed to be an instance of the class rho-theta-position
;;; using change-class:

(setq p1 (make-instance 'x-y-position :x 2 :y 0))

(change-class p1 'rho-theta-position)

;;; The result is that the instance bound to p1 is now
;;; an instance of the class rho-theta-position.
;;; The update-instance-for-different-class method
;;; performed the initialization of the rho and theta
;;; slots based on the values of the x and y slots,
;;; which were maintained by the old instance.
```

After completing all other actions, `change-class` invokes the generic function `update-instance-for-different-class`. The generic function `update-instance-for-different-class` can be used to assign values to slots in the transformed instance.

The generic function `change-class` has several semantic difficulties. First, it performs a destructive operation that can be invoked within a method on an instance that was used to select that method. When multiple methods are involved because methods are being combined, the methods currently executing or about to be executed may no longer be applicable. Second, some implementations might use compiler optimizations of slot access, and when the class of an instance is changed the assumptions the compiler made might be violated. This implies that a programmer must not

use `change-class` inside a method if any methods for that generic function access any slots, or the results are undefined.

See section 27.1.11 as well as `update-instance-for-different-class`.

[Generic function] **class-name** *class*
[Primary method] **class-name** (*class* class)

The generic function `class-name` takes a class object and returns its name. The *class* argument is a class object. The *new-value* argument is any object. The name of the given class is returned.

The name of an anonymous class is `nil`.

If S is a symbol such that $S =$`(class-name C)` and $C =$ `(find-class S)`, then S is the proper name of C (see section 27.1.2).

See also section 27.1.2 and `find-class`.

[Generic function] **(setf** *class-name) new-value class*
[Primary method] **(setf** *class-name) new-value* (*class* class)

The generic function `(setf class-name)` takes a class object and sets its name. The *class* argument is a class object. The *new-value* argument is any object.

[Function] **class-of** *object*

The function `class-of` returns the class of which the given object is an instance. The argument to `class-of` may be any Common Lisp object. The function `class-of` returns the class of which the argument is an instance.

[Function] **compute-applicable-methods** *generic-function function-arguments*

Given a generic function and a set of arguments, the function `compute-applicable-methods` returns the set of methods that are applicable for those arguments.

The methods are sorted according to precedence order. See section 27.1.7.

The *generic-function* argument must be a generic function object. The *function-arguments* argument is a list of the arguments to that generic function. The result is a list of the applicable methods in order of precedence. See section 27.1.7.

[Макрос] **defclass** class-name ({superclass-name}*)
({slot-specifier}*) [[↓class-option]]

class-name ::= *symbol*
superclass-name ::= *symbol*
slot-specifier ::= *slot-name* | (*slot-name* [[↓slot-option]])
slot-name ::= *symbol*
slot-option ::= {*:reader* reader-function-name}*
| {*:writer* writer-function-name}*
| {*:accessor* reader-function-name}*
| {*:allocation* allocation-type}
| {*:initarg* initarg-name}*
| {*:initform* form}
| {*:type* type-specifier}
| {*:documentation* string}

reader-function-name ::= *symbol*
writer-function-name ::= *function-name*
function-name ::= {symbol / `(setf symbol)`}
initarg-name ::= *symbol*
allocation-type ::= `:instance | :class`
class-option ::= `(:default-initargs` *initarg-list*`)`
| `(:documentation` *string*`)`
| `(:metaclass` *class-name*`)`
initarg-list ::= {initarg-name default-initial-value-form}*

The macro `defclass` defines a new named class. It returns the new class object as its result.

The syntax of `defclass` provides options for specifying initialization arguments for slots, for specifying default initialization values for slots, and for requesting that methods on specified generic functions be automatically generated for reading and writing the values of slots. No reader or writer functions are defined by default; their generation must be explicitly requested.

Defining a new class also causes a type of the same name to be defined. The predicate `(typep` *object class-name*`)` returns true if the class of the given object is *class-name* itself or a subclass of the class *class-name*. A class object can be used as a type specifier. Thus `(typep` *object class*`)` returns true if the class of the *object* is *class* itself or a subclass of *class*.

The *class-name* argument is a non-`nil` symbol. It becomes the proper name of the new class. If a class with the same proper name already exists and that class is an instance of `standard-class`, and if the `defclass` form for the definition of the new class specifies a class of class `standard-class`, the definition of the existing class is replaced.

Each *superclass-name* argument is a non-`nil` symbol that specifies a direct superclass of the new class. The new class will inherit slots and methods from each of its direct superclasses, from their direct superclasses, and so on. See section 27.1.3 for a discussion of how slots and methods are inherited.

Each *slot-specifier* argument is the name of the slot or a list consisting of the slot name followed by zero or more slot options. The *slot-name* argument is a symbol that is syntactically valid for use as a variable name. If there are any duplicate slot names, an error is signaled.

The following slot options are available:

- The `:reader` slot option specifies that an unqualified method is to be

defined on the generic function named *reader-function-name* to read the value of the given slot. The *reader-function-name* argument is a non-`nil` symbol. The `:reader` slot option may be specified more than once for a given slot.

- The `:writer` slot option specifies that an unqualified method is to be defined on the generic function named *writer-function-name* to write the value of the slot. The *writer-function-name* argument is a function-name. The `:writer` slot option may be specified more than once for a given slot.

- The `:accessor` slot option specifies that an unqualified method is to be defined on the generic function named *reader-function-name* to read the value of the given slot and that an unqualified method is to be defined on the generic function named `(setf` *reader-function-name*`)` to be used with `setf` to modify the value of the slot. The *reader-function-name* argument is a non-`nil` symbol. The `:accessor` slot option may be specified more than once for a given slot.

- The `:allocation` slot option is used to specify where storage is to be allocated for the given slot. Storage for a slot may be located in each instance or in the class object itself, for example. The value of the *allocation-type* argument can be either the keyword `:instance` or the keyword `:class`. The `:allocation` slot option may be specified at most once for a given slot. If the `:allocation` slot option is not specified, the effect is the same as specifying `:allocation :instance`.

  - If *allocation-type* is `:instance`, a local slot of the given name is allocated in each instance of the class.
  - If *allocation-type* is `:class`, a shared slot of the given name is allocated. The value of the slot is shared by all instances of the class. If a class C_1 defines such a shared slot, any subclass C_2 of C_1 will share this single slot unless the `defclass` form for C_2 specifies a slot of the same name or there is a superclass of C_2 that precedes C_1 in the class precedence list of C_2 and that defines a slot of the same name.

- The `:initform` slot option is used to provide a default initial value form to be used in the initialization of the slot. The `:initform` slot option may be specified at most once for a given slot. This form is evaluated every time it is used to initialize the slot. The lexical environment in which this form is evaluated is the lexical environment in which the `defclass` form was evaluated. Note that the lexical environment refers both to variables and to functions. For local slots, the dynamic environment is the dynamic environment in which `make-instance` was called; for shared slots, the dynamic environment is the dynamic environment in which the `defclass` form was evaluated. See section 27.1.9.

  No implementation is permitted to extend the syntax of `defclass` to allow (*slot-name form*) as an abbreviation for (*slot-name* `:initform` *form*).

- The `:initarg` slot option declares an initialization argument named *initarg-name* and specifies that this initialization argument initializes the given slot. If the initialization argument has a value in the call to `initialize-instance`, the value will be stored into the given slot, and the slot's `:initform` slot option, if any, is not evaluated. If none of the initialization arguments specified for a given slot has a value, the slot is initialized according to the `:initform` slot option, if specified. The `:initarg` slot option can be specified more than once for a given slot. The *initarg-name* argument can be any symbol.

- The `:type` slot option specifies that the contents of the slot will always be of the specified data type. It effectively declares the result type of the reader generic function when applied to an object of this class. The result of attempting to store in a slot a value that does not satisfy the type of the slot is undefined. The `:type` slot option may be specified at most once for a given slot. The `:type` slot option is further discussed in section 27.1.3.

- The `:documentation` slot option provides a documentation string for the slot.

Each class option is an option that refers to the class as a whole or to all class slots. The following class options are available:

- The `:default-initargs` class option is followed by a list of alternating initialization argument names and default initial value forms. If any of these initialization arguments does not appear in the initialization argument list supplied to `make-instance`, the corresponding default initial value form is evaluated, and the initialization argument name and the form's value are added to the end of the initialization argument list before the instance is created (see section 27.1.9). The default initial value form is evaluated each time it is used. The lexical environment in which this form is evaluated is the lexical environment in which the `defclass` form was evaluated. The dynamic environment is the dynamic environment in which `make-instance` was called. If an initialization argument name appears more than once in a `:default-initargs` class option, an error is signaled. The `:default-initargs` class option may be specified at most once.

- The `:documentation` class option causes a documentation string to be attached to the class name. The documentation type for this string is `type`. The form `(documentation` *`class-name`* `'type)` may be used to retrieve the documentation string. The `:documentation` class option may be specified at most once.

- The `:metaclass` class option is used to specify that instances of the class being defined are to have a different metaclass than the default provided by the system (the class `standard-class`). The *class-name* argument is the name of the desired metaclass. The `:metaclass` class option may be specified at most once.

The new class object is returned as the result.

If a class with the same proper name already exists and that class is an instance of `standard-class`, and if the `defclass` form for the definition of the new class specifies a class of class `standard-class`, the existing class is redefined, and instances of it (and its subclasses) are updated to the new definition at the time that they are next accessed (see section 27.1.10).

Note the following rules of `defclass` for standard classes:

- It is not required that the superclasses of a class be defined before the `defclass` form for that class is evaluated.

- All the superclasses of a class must be defined before an instance of the class can be made.
- A class must be defined before it can be used as a parameter specializer in a `defmethod` form.

The Объектная система may be extended to cover situations where these rules are not obeyed.

Some slot options are inherited by a class from its superclasses, and some can be shadowed or altered by providing a local slot description. No class options except `:default-initargs` are inherited. For a detailed description of how slots and slot options are inherited, see section 27.1.3.

The options to `defclass` can be extended. An implementation must signal an error if it observes a class option or a slot option that is not implemented locally.

It is valid to specify more than one reader, writer, accessor, or initialization argument for a slot. No other slot option may appear more than once in a single slot description, or an error is signaled.

If no reader, writer, or accessor is specified for a slot, the slot can be accessed only by the function `slot-value`.

See sections 27.1.2, 27.1.3, 27.1.10, 27.1.5, 27.1.9 as well as `slot-value`, `make-instance`, and `initialize-instance`.

[Макрос] **defgeneric** function-name lambda-list
[[↓option | {method-description}*]]

function-name ::= {symbol / *(setf* `symbol`*)*}
lambda-list ::= `(`{*var*}*
[*&optional* {*var* / *(*`var`*)*}*]
[*&rest* var]
[*&key* {*keyword-parameter*}* [*&allow-other-keys*/]`)`

keyword-parameter ::= *var* | `(`{*var* / *(*`keyword var`*)*}`)`
option ::= `(:argument-precedence-order` {*parameter-name*}+`)`
| `(declare` {*declaration*}+`)`
| `(:documentation` *string*`)`
| `(:method-combination` *symbol* {*arg*}*`)`

| (:generic-function-class *class-name*)
| (:method-class *class-name*)
method-description ::= (:method {*method-qualifier*}*
specialized-lambda-list
[[*{declaration}* | documentation*]]
{*form*}*)
method-qualifier ::= *non-nil-atom*
specialized-lambda-list ::=
({*var* | *(var parameter-specializer-name)* }*
[*&optional {var | (var [initform [supplied-p-parameter]])}**]
[*&rest* var]
[*&key {specialized-keyword-parameter}* [&allow-other-keys]*]
[*&aux {var | (*var *[initform])}**])
specialized-keyword-parameter ::=
var | ({*var* | *(*keyword var*)*} [*initform [supplied-p-parameter]*])
parameter-specializer-name ::= *symbol* | (eql *eql-specializer-form*)

The macro defgeneric is used to define a generic function or to specify options and declarations that pertain to a generic function as a whole.

If (fboundp *function-name*) is nil, a new generic function is created. If (fdefinition *function-specifier*) is a generic function, that generic function is modified. If *function-name*/ names a non-generic function, a macro, or a special operator, an error is signaled.

[X3J13 voted in March 1989 to use fdefinition in the previous paragraph, as shown, rather than symbol-function, as it appeared in the original report on CLOS [5, 7]. The vote also changed all occurrences of *function-specifier* in the original report to *function-name*; this change is reflected here.—GLS]

Each *method-description* defines a method on the generic function. The lambda-list of each method must be congruent with the lambda-list specified by the *lambda-list* option. If this condition does not hold, an error is signaled. See section 27.1.6 for a definition of congruence in this context.

The macro defgeneric returns the generic function object as its result.

The *function-name* argument is a non-nil symbol or a list of the form (setf *symbol*).

The *lambda-list* argument is an ordinary function lambda-list with the following exceptions:

- The use of `&aux` is not allowed.
- Optional and keyword arguments may not have default initial value forms nor use supplied-p parameters. The generic function passes to the method all the argument values passed to it, and only those; default values are not supported. Note that optional and keyword arguments in method definitions, however, can have default initial value forms and can use supplied-p parameters.

The following options are provided. A given option may occur only once, or an error is signaled.

- The `:argument-precedence-order` option is used to specify the order in which the required arguments in a call to the generic function are tested for specificity when selecting a particular method. Each required argument, as specified in the *lambda-list* argument, must be included exactly once as a *parameter-name* so that the full and unambiguous precedence order is supplied. If this condition is not met, an error is signaled.
- The `declare` option is used to specify declarations that pertain to the generic function. The following standard Common Lisp declaration is allowed:
  - An `optimize` declaration specifies whether method selection should be optimized for speed or space, but it has no effect on methods. To control how a method is optimized, an `optimize` declaration must be placed directly in the `defmethod` form or method description. The optimization qualities `speed` and `space` are the only qualities this standard requires, but an implementation can extend the Объектная система Common Lisp'a to recognize other qualities. A simple implementation that has only one method selection technique and ignores the `optimize` declaration is valid.

The `special`, `ftype`, `function`, `inline`, `notinline`, and `declaration` declarations are not permitted. Individual implementations can extend the `declare` option to support additional declarations. If an implementation notices a declaration that it does not support and that has not been proclaimed as a non-standard declaration name in a `declaration` proclamation, it should issue a warning.

- The `:documentation` argument associates a documentation string with the generic function. The documentation type for this string is `function`. The form `(documentation` *function-name/* `'function)` may be used to retrieve this string.

- The `:generic-function-class` option may be used to specify that the generic function is to have a different class than the default provided by the system (the class `standard-generic-function`). The *class-name* argument is the name of a class that can be the class of a generic function. If *function-name* specifies an existing generic function that has a different value for the `:generic-function-class` argument and the new generic function class is compatible with the old, `change-class` is called to change the class of the generic function; otherwise an error is signaled.

- The `:method-class` option is used to specify that all methods on this generic function are to have a different class from the default provided by the system (the class `standard-method`). The *class-name* argument is the name of a class that is capable of being the class of a method.

- The `:method-combination` option is followed by a symbol that names a type of method combination. The arguments (if any) that follow that symbol depend on the type of method combination. Note that the standard method combination type does not support any arguments. However, all types of method combination defined by the short form of `define-method-combination` accept an optional argument named *order*, defaulting to `:most-specific-first`, where a value of `:most-specific-last` reverses the order of the primary methods without affecting the order of the auxiliary methods.

The *method-description* arguments define methods that will be associated with the generic function. The *method-qualifier* and *specialized-lambda-list* arguments in a method description are the same as for `defmethod`.

The *form* arguments specify the method body. The body of the method is enclosed in an implicit block. If *function-name* is a symbol, this block bears the same name as the generic function. If *function-name* is a list of the form `(setf` *`symbol`*`)`, the name of the block is *symbol*.

The generic function object is returned as the result.

The effect of the `defgeneric` macro is as if the following three steps were performed: first, methods defined by previous `defgeneric` forms are removed; second, `ensure-generic-function` is called; and finally, methods specified by the current `defgeneric` form are added to the generic function.

If no method descriptions are specified and a generic function of the same name does not already exist, a generic function with no methods is created.

The *lambda-list* argument of `defgeneric` specifies the shape of lambda-lists for the methods on this generic function. All methods on the resulting generic function must have lambda-lists that are congruent with this shape. If a `defgeneric` form is evaluated and some methods for that generic function have lambda-lists that are not congruent with that given in the `defgeneric` form, an error is signaled. For further details on method congruence, see section 27.1.6.

Implementations can extend `defgeneric` to include other options. It is required that an implementation signal an error if it observes an option that is not implemented locally.

See section 27.1.6 as well as `defmethod`, `ensure-generic-function`, and `generic-function`.

[Макрос] **define-method-combination** name [[↓short-form-option]]
[Макрос] **define-method-combination** name lambda-list
({method-group-specifier}*)
[(`:arguments` . lambda-list)]
[(`:generic-function` generic-fn-symbol)]
[[{declaration}* | doc-string]]
{form}*

short-form-option ::= `:documentation` *`string`*
| `:identity-with-one-argument` *`boolean`*
| `:operator` *`operator`*
method-group-specifier ::= `(`*variable* { *{qualifier-pattern}+* | *predicate*}
[[↓*long-form-option*]] `)`

long-form-option ::= `:description` *`format-string`*
| `:order` *`order`*
| `:required` *`boolean`*

The macro `define-method-combination` is used to define new types of method combination.

There are two forms of `define-method-combination`. The short form is a simple facility for the cases that are expected to be most commonly needed. The long form is more powerful but more verbose. It resembles `defmacro` in that the body is an expression, usually using backquote, that computes a Lisp form. Thus arbitrary control structures can be implemented. The long form also allows arbitrary processing of method qualifiers.

In both the short and long forms, *name* is a symbol. By convention, non-keyword, non-`nil` symbols are usually used.

The short-form syntax of `define-method-combination` is recognized when the second subform is a non-`nil` symbol or is not present. When the short form is used, *name* is defined as a type of method combination that produces a Lisp form (*`operator method-call method-call ...`*). The *operator* is a symbol that can be the name of a function, macro, or special operator. The *operator* can be specified by a keyword option; it defaults to *name*.

Keyword options for the short form are the following:

- The `:documentation` option is used to document the method-combination type.
- The `:identity-with-one-argument` option enables an optimization when *boolean* is true (the default is false). If there is exactly one applicable method and it is a primary method, that method serves as the effective method and *operator* is not called. This optimization avoids the need to create a new effective method and avoids the overhead of a function call. This option is designed to be used with operators such as `progn`, `and`, `+`, and `max`.
- The `:operator` option specifies the name of the operator. The *operator* argument is a symbol that can be the name of a function, macro, or special operator. By convention, *name* and *operator* are often the same symbol. This is the default, but it is not required.

None of the subforms is evaluated.

These types of method combination require exactly one qualifier per method. An error is signaled if there are applicable methods with no qualifiers or with qualifiers that are not supported by the method combination type.

A method combination procedure defined in this way recognizes two roles for methods. A method whose one qualifier is the symbol naming this type of method combination is defined to be a primary method. At least one primary method must be applicable or an error is signaled. A method with `:around` as its one qualifier is an auxiliary method that behaves the same as an `:around` method in standard method combination. The function `call-next-method` can be used only in `:around` methods; it cannot be used in primary methods defined by the short form of the `define-method-combination` macro.

A method combination procedure defined in this way accepts an optional argument named *order*, which defaults to `:most-specific-first`. A value of `:most-specific-last` reverses the order of the primary methods without affecting the order of the auxiliary methods.

The short form automatically includes error checking and support for `:around` methods.

For a discussion of built-in method combination types, see section 27.1.7.

The long-form syntax of `define-method-combination` is recognized when the second subform is a list.

The *lambda-list* argument is an ordinary lambda-list. It receives any arguments provided after the name of the method combination type in the `:method-combination` option to `defgeneric`.

A list of method group specifiers follows. Each specifier selects a subset of the applicable methods to play a particular role, either by matching their qualifiers against some patterns or by testing their qualifiers with a predicate. These method group specifiers define all method qualifiers that can be used with this type of method combination. If an applicable method does not fall into any method group, the system signals the error that the method is invalid for the kind of method combination in use.

Each method group specifier names a variable. During the execution of the forms in the body of `define-method-combination`, this variable is bound to a list of the methods in the method group. The methods in this list occur in most-specific-first order.

A qualifier pattern is a list or the symbol `*`. A method matches a qualifier

pattern if the method's list of qualifiers is `equal` to the qualifier pattern (except that the symbol `*` in a qualifier pattern matches anything). Thus a qualifier pattern can be one of the following: the empty list `()`, which matches unqualified methods; the symbol `*`, which matches all methods; a true list, which matches methods with the same number of qualifiers as the length of the list when each qualifier matches the corresponding list element; or a dotted list that ends in the symbol `*` (the `*` matches any number of additional qualifiers).

Each applicable method is tested against the qualifier patterns and predicates in left-to-right order. As soon as a qualifier pattern matches or a predicate returns true, the method becomes a member of the corresponding method group and no further tests are made. Thus if a method could be a member of more than one method group, it joins only the first such group. If a method group has more than one qualifier pattern, a method need only satisfy one of the qualifier patterns to be a member of the group.

The name of a predicate function can appear instead of qualifier patterns in a method group specifier. The predicate is called for each method that has not been assigned to an earlier method group; it is called with one argument, the method's qualifier list. The predicate should return true if the method is to be a member of the method group. A predicate can be distinguished from a qualifier pattern because it is a symbol other than `nil` or `*`.

If there is an applicable method whose qualifiers are not valid for the method combination type, the function `invalid-method-error` is called.

Method group specifiers can have keyword options following the qualifier patterns or predicate. Keyword options can be distinguished from additional qualifier patterns because they are neither lists nor the symbol `*`. The keyword options are:

- The `:description` option is used to provide a description of the role of methods in the method group. Programming environment tools use `(apply #'format stream` *`format-string`* `(method-qualifiers` *`method`*`))` to print this description, which is expected to be concise. This keyword option allows the description of a method qualifier to be defined in the same module that defines the meaning of the method qualifier. In most cases, *format-string* will not contain any `format` directives, but they are available for generality. If `:description` is not specified, a default description is generated based on the variable name and the qualifier patterns and on whether this method group includes

the unqualified methods. The argument *format-string* is not evaluated.

- The `:order` option specifies the order of methods. The *order* argument is a form that evaluates to `:most-specific-first` or `:most-specific-last`. If it evaluates to any other value, an error is signaled. This keyword option is a convenience and does not add any expressive power. If `:order` is not specified, it defaults to `:most-specific-first`.

- The `:required` option specifies whether at least one method in this method group is required. If the *boolean* argument is non-`nil` and the method group is empty (that is, no applicable methods match the qualifier patterns or satisfy the predicate), an error is signaled. This keyword option is a convenience and does not add any expressive power. If `:required` is not specified, it defaults to `nil`. The *boolean* argument is not evaluated.

The use of method group specifiers provides a convenient syntax to select methods, to divide them among the possible roles, and to perform the necessary error checking. It is possible to perform further filtering of methods in the body forms by using normal list-processing operations and the functions `method-qualifiers` and `invalid-method-error`. It is permissible to use `setq` on the variables named in the method group specifiers and to bind additional variables. It is also possible to bypass the method group specifier mechanism and do everything in the body forms. This is accomplished by writing a single method group with `*` as its only qualifier pattern; the variable is then bound to a list of all of the applicable methods, in most-specific-first order.

The body *forms* compute and return the Lisp form that specifies how the methods are combined, that is, the effective method. The effective method uses the macro `call-method`. The definition of this macro has lexical scope and is available only in an effective method form. Given a method object in one of the lists produced by the method group specifiers and a list of next methods, the macro `call-method` will invoke the method so that `call-next-method` will have available the next methods.

When an effective method has no effect other than to call a single method, some implementations employ an optimization that uses the single method directly as the effective method, thus avoiding the need to create a new effective method. This optimization is active when the effective method

form consists entirely of an invocation of the `call-method` macro whose first subform is a method object and whose second subform is `nil`. Each `define-method-combination` body is responsible for stripping off redundant invocations of `progn`, `and`, `multiple-value-prog1`, and the like, if this optimization is desired.

The list (`:arguments` . *lambda-list*) can appear before any declaration or documentation string. This form is useful when the method combination type performs some specific behavior as part of the combined method and that behavior needs access to the arguments to the generic function. Each parameter variable defined by *lambda-list* is bound to a form that can be inserted into the effective method. When this form is evaluated during execution of the effective method, its value is the corresponding argument to the generic function. If *lambda-list* is not congruent to the generic function's lambda-list, additional ignored parameters are automatically inserted until it is congruent. Thus it is permissible for *lambda-list* to receive fewer arguments than the number that the generic function expects.

Erroneous conditions detected by the body should be reported with `method-combination-error` or `invalid-method-error`; these functions add any necessary contextual information to the error message and will signal the appropriate error.

The body *forms* are evaluated inside the bindings created by the lambda-list and method group specifiers. Declarations at the head of the body are positioned directly inside bindings created by the lambda-list and outside the bindings of the method group variables. Thus method group variables cannot be declared.

Within the body *forms*, *generic-function-symbol* is bound to the generic function object.

If a *doc-string* argument is present, it provides the documentation for the method combination type.

The functions `method-combination-error` and `invalid-method-error` can be called from the body *forms* or from functions called by the body *forms*. The actions of these two functions can depend on implementation-dependent dynamic variables automatically bound before the generic function `compute-effective-method` is called.

Note that two methods with identical specializers, but with different qualifiers, are not ordered by the algorithm described in step 2 of the method selection and combination process described in section 27.1.7. Normally the two methods play different roles in the effective method because they have

different qualifiers, and no matter how they are ordered in the result of step 2 the effective method is the same. If the two methods play the same role and their order matters, an error is signaled. This happens as part of the qualifier pattern matching in `define-method-combination`.

The value returned by the `define-method-combination` macro is the new method combination object.

Most examples of the long form of `define-method-combination` also illustrate the use of the related functions that are provided as part of the declarative method combination facility.

```
;;; Examples of the short form of define-method-combination

(define-method-combination and :identity-with-one-argument t)

(defmethod func and ((x class1) y)
  ...)

;;; The equivalent of this example in the long form is:

(define-method-combination and
        (&optional (order ':most-specific-first))
        ((around (:around))
         (primary (and) :order order :required t))
  (let ((form (if (rest primary)
                  `(and ,@(mapcar #'(lambda (method)
                                      `(call-method ,method ()))
                                  primary))
                  `(call-method ,(first primary) ()))))
    (if around
        `(call-method ,(first around)
                      (,@(rest around)
                       (make-method form)))
        form)))
```

```
;;; Examples of the long form of define-method-combination

;;; The default method-combination technique

(define-method-combination standard ()
        ((around (:around))
         (before (:before))
         (primary () :required t)
         (after (:after)))
  (flet ((call-methods (methods)
           (mapcar #'(lambda (method)
                       `(call-method ,method ()))
                   methods)))
    (let ((form (if (or before after (rest primary))
                    `(multiple-value-prog1
                       (progn ,@(call-methods before)
                              (call-method ,(first primary)
                                           ,(rest primary)))
                       ,@(call-methods (reverse after)))
                    `(call-method ,(first primary) ()))))
      (if around
          `(call-method ,(first around)
                        (,@(rest around)
                         (make-method form)))
          form))))

;;; A simple way to try several methods until one returns non-nil

(define-method-combination or ()
        ((methods (or)))
  `(or ,@(mapcar #'(lambda (method)
                     `(call-method ,method ()))
                 methods)))
```

```
;;; A more complete version of the preceding

(define-method-combination or
        (&optional (order ':most-specific-first))
        ((around (:around))
         (primary (or)))
  ;; Process the order argument
  (case order
    (:most-specific-first)
    (:most-specific-last (setq primary (reverse primary)))
    (otherwise (method-combination-error
                "~S is an invalid order.~@
                 :most-specific-first and :most-specific-last ~
                   are the possible values."
                                order)))
  ;; Must have a primary method
  (unless primary
    (method-combination-error "A primary method is required."))
  ;; Construct the form that calls the primary methods
  (let ((form (if (rest primary)
                  `(or ,@(mapcar #'(lambda (method)
                                     `(call-method ,method ()))
                                 primary))
                  `(call-method ,(first primary) ()))))
    ;; Wrap the around methods around that form
    (if around
        `(call-method ,(first around)
                      (,@(rest around)
                       (make-method form)))
        form)))



;;; The same thing, using the :order and :required keyword options
(define-method-combination or
        (&optional (order ':most-specific-first))
        ((around (:around))
         (primary (or) :order order :required t))
```

```
  (let ((form (if (rest primary)
                  `(or ,@(mapcar #'(lambda (method)
                                     `(call-method ,method ()))
                                 primary))
                  `(call-method ,(first primary) ()))))
    (if around
        `(call-method ,(first around)
                      (,@(rest around)
                       (make-method form)))
        form)))

;;; This short-form call is behaviorally identical to the preceding.
(define-method-combination or :identity-with-one-argument t)

;;; Order methods by positive integer qualifiers; note that :around
;;; methods are disallowed here in order to keep the example small.

(define-method-combination example-method-combination ()
        ((methods positive-integer-qualifier-p))
  `(progn ,@(mapcar #'(lambda (method)
                        `(call-method ,method ()))
                    (stable-sort methods #'<
                      :key #'(lambda (method)
                               (first (method-qualifiers
                                       method)))))))

(defun positive-integer-qualifier-p (method-qualifiers)
  (and (= (length method-qualifiers) 1)
       (typep (first method-qualifiers) '(integer 0 *))))

;;; Example of the use of :arguments
(define-method-combination progn-with-lock ()
        ((methods ()))
        (:arguments object)
```

```
`(unwind-protect
    (progn (lock (object-lock ,object))
           ,@(mapcar #'(lambda (method)
                         `(call-method ,method ()))
                     methods))
   (unlock (object-lock ,object))))
```

The `:method-combination` option of `defgeneric` is used to specify that a generic function should use a particular method combination type. The argument to the `:method-combination` option is the name of a method combination type.

See sections 27.1.7 and 27.1.7 as well as `call-method`, `method-qualifiers`, `method-combination-error`, `invalid-method-error`, and `defgeneric`.

[Макрос] **defmethod** function-name {method-qualifier}*
specialized-lambda-list
[[{declaration}* | doc-string]] {form}*

function-name ::= {symbol / *(setf* `symbol`*)*}
method-qualifier ::= *non-nil-atom*
parameter-specializer-name ::= *symbol* | `(eql` *eql-specializer-form*`)`

The macro `defmethod` defines a method on a generic function.

If `(fboundp` *function-name*`)` is `nil`, a generic function is created with default values for the argument precedence order (each argument is more specific than the arguments to its right in the argument list), for the generic function class (the class `standard-generic-function`), for the method class (the class `standard-method`), and for the method combination type (the standard method combination type). The lambda-list of the generic function is congruent with the lambda-list of the method being defined; if the `defmethod` form mentions keyword arguments, the lambda-list of the generic function will mention `&key` (but no keyword arguments). If *function-name* names a non-generic function, a macro, or a special operator, an error is signaled.

If a generic function is currently named by *function-name*, where *function-name* is a symbol or a list of the form `(setf symbol)`, the lambda-list of the method must be congruent with the lambda-list of the generic function. If this condition does not hold, an error is signaled. See section 27.1.6 for a definition of congruence in this context.

The *function-name* argument is a non-`nil` symbol or a list of the form `(setf symbol)`. It names the generic function on which the method is defined.

Each *method-qualifier* argument is an object that is used by method combination to identify the given method. A method qualifier is a non-`nil` atom. The method combination type may further restrict what a method qualifier may be. The standard method combination type allows for unqualified methods or methods whose sole qualifier is the keyword `:before`, the keyword `:after`, or the keyword `:around`.

A *specialized-lambda-list* is like an ordinary function lambda-list except that the name of a required parameter can be replaced by a specialized parameter, a list of the form `(variable-name parameter-specializer-name)`. Only required parameters may be specialized. A parameter specializer name is a symbol that names a class or `(eql eql-specializer-form)`. The parameter specializer name `(eql eql-specializer-form)` indicates that the corresponding argument must be `eql` to the object that is the value of *eql-specializer-form* for the method to be applicable. If no parameter specializer name is specified for a given required parameter, the parameter specializer defaults to the class named `t`. See section 27.1.6.

The *form* arguments specify the method body. The body of the method is enclosed in an implicit block. If *function-name* is a symbol, this block bears the same name as the generic function. If *function-name* is a list of the form `(setf symbol)`, the name of the block is *symbol*.

The result of `defmethod` is the method object.

The class of the method object that is created is that given by the method class option of the generic function on which the method is defined.

If the generic function already has a method that agrees with the method being defined on parameter specializers and qualifiers, `defmethod` replaces the existing method with the one now being defined. See section 27.1.6 for a definition of agreement in this context.

The parameter specializers are derived from the parameter specializer names as described in section 27.1.6.

The expansion of the `defmethod` macro refers to each specialized parameter (see the `ignore` declaration specifier), including parameters that have an explicit parameter specializer name of `t`. This means that a compiler warning does not occur if the body of the method does not refer to a specialized parameter. Note that a parameter that specializes on `t` is not synonymous with an unspecialized parameter in this context.

See sections 27.1.6, 27.1.6, and 27.1.6.

[At this point the original CLOS report [5, 7] contained a specification for `describe` as a generic function. This specification is omitted here because X3J13 voted in March 1989 not to make `describe` a generic function after all (see `describe-object`).—GLS]

[Generic function] **documentation** *x* **&optional** *doc-type*
[Primary method] **documentation** (*method* standard-method) **&optional** *doc-type*
[Primary method] **documentation** (*generic-function* standard-generic-function) **&optional** *doc-type*
[Primary method] **documentation** (*class* standard-class) **&optional** *doc-type*
[Primary method] **documentation** (*method-combination* method-combination) **&optional** *doc-type*
[Primary method] **documentation** (*slot-description* standard-slot-description) **&optional** *doc-type*
[Primary method] **documentation** (*symbol* symbol) **&optional** *doc-type*
[Primary method] **documentation** (*list* list) **&optional** *doc-type*

The ordinary function `documentation` (see section **??**) is replaced by a generic function. The generic function `documentation` returns the documentation string associated with the given object if it is available; otherwise `documentation` returns `nil`.

The first argument of `documentation` is a symbol, a function-name list of the form `(setf` *`symbol`*`)`, a method object, a class object, a generic function object, a method combination object, or a slot description object. Whether a second argument should be supplied depends on the type of the first argument.

- If the first argument is a method object, a class object, a generic function object, a method combination object, or a slot description object, the second argument must not be supplied, or an error is signaled.

- If the first argument is a symbol or a list of the form `(setf` *`symbol`*`)`, the second argument must be supplied.

  - The forms

    (documentation *symbol* 'function)

    and

    (documentation '(setf *symbol*) 'function)

    return the documentation string of the function, generic function, special operator, or macro named by the symbol or list.
  - The form `(documentation` *`symbol`* `'variable)` returns the documentation string of the special variable or constant named by the symbol.
  - The form `(documentation` *`symbol`* `'structure)` returns the documentation string of the `defstruct` structure named by the symbol.
  - The form `(documentation` *`symbol`* `'type)` returns the documentation string of the class object named by the symbol, if there is such a class. If there is no such class, it returns the documentation string of the type specifier named by the symbol.
  - The form `(documentation` *`symbol`* `'setf)` returns the documentation string of the `defsetf` or `define-setf-method` definition associated with the symbol.
  - The form `(documentation` *`symbol`* `'method-combination)` returns the documentation string of the method combination type named by the symbol.

An implementation may extend the set of symbols that are acceptable as the second argument. If a symbol is not recognized as an acceptable argument by the implementation, an error must be signaled.

The documentation string associated with the given object is returned unless none is available, in which case `documentation` returns `nil`.

[Generic function] **(setf** *documentation) new-value x* **&optional** *doc-type*
[Primary method] **(setf** *documentation) new-value*
(*method* standard-method) **&optional** *doc-type*
[Primary method] **(setf** *documentation) new-value*
(*generic-function* standard-generic-function) **&optional** *doc-type*
[Primary method] **(setf** *documentation) new-value* (*class* standard-class) **&optional** *doc-type*
[Primary method] **(setf** *documentation) new-value*
(*method-combination* method-combination) **&optional** *doc-type*
[Primary method] **(setf** *documentation) new-value*
(*slot-description* standard-slot-description) **&optional** *doc-type*
[Primary method] **(setf** *documentation) new-value* (*symbol* symbol) **&optional** *doc-type*
[Primary method] **(setf** *documentation) new-value* (*list* list) **&optional** *doc-type*

The generic function `(setf documentation)` is used to update the documentation.

The first argument of `(setf documentation)` is the new documentation.

The second argument of `documentation` is a symbol, a function-name list of the form `(setf` *`symbol`*`)`, a method object, a class object, a generic function object, a method combination object, or a slot description object. Whether a third argument should be supplied depends on the type of the second argument. See `documentation`.

[Function] **ensure-generic-function** *function-name* **&key** *:lambda-list :argument-precedence-order :declare :documentation :generic-function-class :method-combination :method-class :environment*

function-name ::= {symbol / *`(setf`* `symbol`*`)`*}

The function `ensure-generic-function` is used to define a globally named generic function with no methods or to specify or modify options and declarations that pertain to a globally named generic function as a whole.

If `(fboundp` *`function-name`*`)` is `nil`, a new generic function is created. If `(fdefinition` *function-name*`)` is a non-generic function, a macro, or a special operator, an error is signaled.

[X3J13 voted in March 1989 to use `fdefinition` in the previous paragraph, as shown, rather than `symbol-function`, as it appeared in the original report on CLOS [5, 7]. The vote also changed all occurrences of *function-specifier* in the original report to *function-name*; this change is reflected here.—GLS]

If *function-name* specifies a generic function that has a different value for any of the following arguments, the generic function is modified to have the new value: `:argument-precedence-order`, `:declare`, `:documentation`, `:method-combination`.

If *function-name* specifies a generic function that has a different value for the `:lambda-list` argument, and the new value is congruent with the lambda-lists of all existing methods or there are no methods, the value is changed; otherwise an error is signaled.

If *function-name* specifies a generic function that has a different value for the `:generic-function-class` argument and if the new generic function class is compatible with the old, `change-class` is called to change the class of the generic function; otherwise an error is signaled.

If *function-name* specifies a generic function that has a different `:method-class` value, the value is changed but any existing methods are not changed.

The *function-name* argument is a symbol or a list of the form `(setf` *`symbol`*`)`.

The keyword arguments correspond to the *option* arguments of `defgeneric`, except that the `:method-class` and `:generic-function-class` arguments can be class objects as well as names.

The `:environment` argument is the same as the `&environment` argument to macro expansion functions. It is typically used to distinguish between compile-time and run-time environments.

The `:method-combination` argument is a method combination object.

The generic function object is returned. See `defgeneric`.

[Function] **find-class** *symbol* **&optional** *errorp environment*

The function `find-class` returns the class object named by the given symbol in the given environment.

The first argument to `find-class` is a symbol.

If there is no such class and the *errorp* argument is not supplied or is non-`nil`, `find-class` signals an error. If there is no such class and the *errorp* argument is `nil`, `find-class` returns `nil`. The default value of *errorp* is `t`.

The optional *environment* argument is the same as the `&environment` argument to macro expansion functions. It is typically used to distinguish between compile-time and run-time environments.

The result of `find-class` is the class object named by the given symbol.

The class associated with a particular symbol can be changed by using `setf` with `find-class`. The results are undefined if the user attempts to change the class associated with a symbol that is defined as a type specifier in chapter 4. See section 27.1.4.

[Generic function] **find-method** *generic-function method-qualifiers specializers* **&optional** *errorp*
[Primary method] **find-method**
(*generic-function* standard-generic-function) *method-qualifiers specializers* **&optional** *errorp*

The generic function `find-method` takes a generic function and returns the method object that agrees on method qualifiers and parameter specializers with the *method-qualifiers* and *specializers* arguments of `find-method`. See section 27.1.6 for a definition of agreement in this context.

The *generic-function* argument is a generic function.

The *method-qualifiers* argument is a list of the method qualifiers for the method. The order of the method qualifiers is significant.

The *specializers* argument is a list of the parameter specializers for the method. It must correspond in length to the number of required arguments of the generic function, or an error is signaled. This means that to obtain the default method on a given generic function, a list whose elements are the class named `t` must be given.

If there is no such method and the *errorp* argument is not supplied or is non-`nil`, `find-method` signals an error. If there is no such method and the *errorp* argument is `nil`, `find-method` returns `nil`. The default value of *errorp* is `t`.

The result of `find-method` is the method object with the given method qualifiers and parameter specializers.

See section 27.1.6.

[Generic function] **function-keywords** *method*
[Primary method] **function-keywords** (*method* standard-method)

The generic function `function-keywords` is used to return the keyword parameter specifiers for a given method.

The *method* argument is a method object.

The generic function `function-keywords` returns two values: a list of the explicitly named keywords and a boolean that states whether `&allow-other-keys` had been specified in the method definition.

[Специальный оператор] **generic-flet** ({(function-name lambda-list
[[↓option | {method-description}*]])}*)
{form}*

The `generic-flet` special operator is analogous to the `flet` special operator. It produces new generic functions and establishes new lexical function definition bindings. Each generic function is created with the set of methods specified by its method descriptions.

The special operator `generic-flet` is used to define generic functions whose names are meaningful only locally and to execute a series of forms with these function definition bindings. Any number of such local generic functions may be defined.

The names of functions defined by `generic-flet` have lexical scope; they retain their local definitions only within the body of the `generic-flet`. Any references within the body of the `generic-flet` to functions whose names are the same as those defined within the `generic-flet` are thus references to the local functions instead of to any global functions of the same names. The scope of these generic function definition bindings, however, includes only the body of `generic-flet`, not the definitions themselves. Within the method bodies, local function names that match those being defined refer to global functions defined outside the `generic-flet`. It is thus not possible to define recursive functions with `generic-flet`.

The *function-name*, *lambda-list*, *option*, *method-qualifier*, and *specialized-lambda-list* arguments are the same as for `defgeneric`.

A `generic-flet` local method definition is identical in form to the method definition part of a `defmethod`.

The body of each method is enclosed in an implicit block. If *function-name* is a symbol, this block bears the same name as the generic function. If *function-name* is a list of the form `(setf` *`symbol`*`)`, the name of the block

is *symbol*.

The result returned by `generic-flet` is the value or values returned by the last form executed. If no forms are specified, `generic-flet` returns `nil`.

See `generic-labels`, `defmethod`, `defgeneric`, and `generic-function`.

[Макрос] **generic-function** lambda-list [[↓option | {method-description}*]]

option ::= `(:argument-precedence-order {`*parameter-name*`}+)`
| `(declare {`*declaration*`}+)`
| `(:documentation` *string*`)`
| `(:method-combination` *symbol* `{`*arg*`}*)`
| `(:generic-function-class` *class-name*`)`
| `(:method-class` *class-name*`)`

method-description ::= `(:method` {*method-qualifier*}* *specialized-lambda-list* {*declaration* /

The `generic-function` macro creates an anonymous generic function. The generic function is created with the set of methods specified by its method descriptions.

The *option*, *method-qualifier*, and *specialized-lambda-list* arguments are the same as for `defgeneric`.

The generic function object is returned as the result.

If no method descriptions are specified, an anonymous generic function with no methods is created.

See `defgeneric`, `generic-flet`, `generic-labels`, and `defmethod`.

[Специальный оператор] **generic-labels** ({(function-name lambda-list
[[↓option | {method-description}*]])}*)
{form}*

The `generic-labels` special operator is analogous to the `labels` special operator. It produces new generic functions and establishes new lexical function definition bindings. Each generic function is created with the set of methods specified by its method descriptions.

The special operator `generic-labels` is used to define generic functions whose names are meaningful only locally and to execute a series of forms with these function definition bindings. Any number of such local generic functions may be defined.

The names of functions defined by `generic-labels` have lexical scope; they retain their local definitions only within the body of the `generic-labels` construct. Any references within the body of the `generic-labels` construct to functions whose names are the same as those defined within the `generic-labels` form are thus references to the local functions instead of to any global functions of the same names. The scope of these generic function definition bindings includes the method bodies themselves as well as the body of the `generic-labels` construct.

The *function-name*, *lambda-list*, *option*, *method-qualifier*, and *specialized-lambda-list* arguments are the same as for `defgeneric`.

A `generic-labels` local method definition is identical in form to the method definition part of a `defmethod`.

The body of each method is enclosed in an implicit block. If *function-name* is a symbol, this block bears the same name as the generic function. If *function-name* is a list of the form `(setf` *`symbol`*`)`, the name of the block is *symbol*.

The result returned by `generic-labels` is the value or values returned by the last form executed. If no forms are specified, `generic-labels` returns `nil`.

See `generic-flet`, `defmethod`, `defgeneric`, `generic-function`.

[Generic function] **initialize-instance** *instance* **&rest** *initargs*
[Primary method] **initialize-instance** (*instance* standard-object) **&rest** *initargs*

The generic function `initialize-instance` is called by `make-instance` to initialize a newly created instance. The generic function `initialize-instance` is called with the new instance and the defaulted initialization arguments.

The system-supplied primary method on `initialize-instance` initializes the slots of the instance with values according to the initialization arguments and the `:initform` forms of the slots. It does this by calling the generic function `shared-initialize` with the following arguments: the instance, `t` (this indicates that all slots for which no initialization arguments are provided should be initialized according to their `:initform` forms) and the defaulted initialization arguments.

The *instance* argument is the object to be initialized.

The *initargs* argument consists of alternating initialization argument names and values.

The modified instance is returned as the result.

Programmers can define methods for `initialize-instance` to specify actions to be taken when an instance is initialized. If only `:after` methods are defined, they will be run after the system-supplied primary method for initialization and therefore will not interfere with the default behavior of `initialize-instance`.

See sections 27.1.9, 27.1.9, and 27.1.9 as well as `shared-initialize`, `make-instance`, `slot-boundp`, and `slot-makunbound`.

[Function] **invalid-method-error** *method format-string* **&rest** *args*

The function `invalid-method-error` is used to signal an error when there is an applicable method whose qualifiers are not valid for the method combination type. The error message is constructed by using a `format` string and any arguments to it. Because an implementation may need to add additional contextual information to the error message, `invalid-method-error` should be called only within the dynamic extent of a method combination function.

The function `invalid-method-error` is called automatically when a method fails to satisfy every qualifier pattern and predicate in a `define-method-combination` form. A method combination function that imposes additional restrictions should call `invalid-method-error` explicitly if it encounters a method it cannot accept.

The *method* argument is the invalid method object.

The *format-string* argument is a control string that can be given to `format`, and *args* are any arguments required by that string.

Whether `invalid-method-error` returns to its caller or exits via `throw` is implementation-dependent.

See `define-method-combination`.

[Generic function] **make-instance** *class* **&rest** *initargs*
[Primary method] **make-instance** (*class* standard-class) **&rest** *initargs*
[Primary method] **make-instance** (*class* symbol) **&rest** *initargs*

The generic function `make-instance` creates a new instance of the given class.

The generic function `make-instance` may be used as described in section 27.1.9.

The *class* argument is a class object or a symbol that names a class. The remaining arguments form a list of alternating initialization argument names and values.

If the second of the preceding methods is selected, that method invokes `make-instance` on the arguments `(find-class` *`class`*`)` and *initargs*.

The initialization arguments are checked within `make-instance` (see section 27.1.9).

The new instance is returned.

The meta-object protocol can be used to define new methods on `make-instance` to replace the object-creation protocol.

See section 27.1.9 as well as `defclass`, `initialize-instance`, and `class-of`.

[Generic function] **make-instances-obsolete** *class*
[Primary method] **make-instances-obsolete** (*class* standard-class)
[Primary method] **make-instances-obsolete** (*class* symbol)

The generic function `make-instances-obsolete` is invoked automatically by the system when `defclass` has been used to redefine an existing standard class and the set of local slots accessible in an instance is changed or the order of slots in storage is changed. It can also be explicitly invoked by the user.

The function `make-instances-obsolete` has the effect of initiating the process of updating the instances of the class. During updating, the generic function `update-instance-for-redefined-class` will be invoked.

The *class* argument is a class object symbol that names the class whose instances are to be made obsolete.

If the second of the preceding methods is selected, that method invokes `make-instances-obsolete` on `(find-class` *`class`*`)`.

The modified class is returned. The result of `make-instances-obsolete` is `eq` to the *class* argument supplied to the first of the preceding methods.

See section 27.1.10 as well as `update-instance-for-redefined-class`.

[Function] **method-combination-error** *format-string* **&rest** *args*

The function `method-combination-error` is used to signal an error in method combination. The error message is constructed by using a

`format` string and any arguments to it. Because an implementation may need to add additional contextual information to the error message, `method-combination-error` should be called only within the dynamic extent of a method combination function.

The *format-string* argument is a control string that can be given to `format`, and *args* are any arguments required by that string.

Whether `method-combination-error` returns to its caller or exits via `throw` is implementation-dependent.

See `define-method-combination`.

[Generic function] **method-qualifiers** *method*
[Primary method] **method-qualifiers** (*method* standard-method)

The generic function `method-qualifiers` returns a list of the qualifiers of the given method.

The *method* argument is a method object.

A list of the qualifiers of the given method is returned.

Example:

```
(setq methods (remove-duplicates methods
                                 :from-end t
                                 :key #'method-qualifiers
                                 :test #'equal))
```

See `define-method-combination`.

[Function] **next-method-p**

The locally defined function `next-method-p` can be used within the body of a method defined by a method-defining form to determine whether a next method exists.

The function `next-method-p` takes no arguments.

The function `next-method-p` returns true or false.

Like `call-next-method`, the function `next-method-p` has lexical scope (for it is defined only within the body of a method defined by a method-defining form) and indefinite extent.

See `call-next-method`.

[Generic function] **no-applicable-method** *generic-function* **&rest** *function-arguments*
[Primary method] **no-applicable-method** (*generic-function* t) **&rest** *function-arguments*

The generic function `no-applicable-method` is called when a generic function of the class `standard-generic-function` is invoked and no method on that generic function is applicable. The default method signals an error.

The generic function `no-applicable-method` is not intended to be called by programmers. Programmers may write methods for it.

The *generic-function* argument of `no-applicable-method` is the generic function object on which no applicable method was found.

The *function-arguments* argument is a list of the arguments to that generic function.

[Generic function] **no-next-method** *generic-function method* **&rest** *args*
[Primary method] **no-next-method**
(*generic-function* standard-generic-function) (*method* standard-method) **&rest** *args*

The generic function `no-next-method` is called by `call-next-method` when there is no next method. The system-supplied method on `no-next-method` signals an error.

The generic function `no-next-method` is not intended to be called by programmers. Programmers may write methods for it.

The *generic-function* argument is the generic function object to which the method that is the second argument belongs.

The *method* argument is the method that contains the call to `call-next-method` for which there is no next method.

The *args* argument is a list of the arguments to `call-next-method`.

See `call-next-method`.

[Generic function] **print-object** *object stream*
[Primary method] **print-object** (*object* standard-object) *stream*

The generic function `print-object` writes the printed representation of an object to a stream. The function `print-object` is called by the print system; it should not be called by the user.

Each implementation must provide a method on the class `standard-object` and methods on enough other classes so as to ensure that there is always an applicable method. Implementations are free to add methods for other classes. Users can write methods for `print-object` for their own classes if they do not wish to inherit an implementation-supplied method.

The first argument is any Lisp object. The second argument is a stream; it cannot be `t` or `nil`.

The function `print-object` returns its first argument, the object.

Methods on `print-object` must obey the print control special variables named `*print-`*xxx*`*` for various *xxx*. The specific details are the following:

- Each method must implement `*print-escape*`.
- The `*print-pretty*` control variable can be ignored by most methods other than the one for lists.
- The `*print-circle*` control variable is handled by the printer and can be ignored by methods.
- The printer takes care of `*print-level*` automatically, provided that each method handles exactly one level of structure and calls `write` (or an equivalent function) recursively if there are more structural levels. The printer's decision of whether an object has components (and therefore should not be printed when the printing depth is not less than `*print-level*`) is implementation-dependent. In some implementations its `print-object` method is not called; in others the method is called, and the determination that the object has components is based on what it tries to write to the stream.
- Methods that produce output of indefinite length must obey `*print-length*`, but most methods other than the one for lists can ignore it.
- The `*print-base*`, `*print-radix*`, `*print-case*`, `*print-gensym*`, and `*print-array*` control variables apply to specific types of objects and are handled by the methods for those objects.

- X3J13 voted in June 1989 to add the following point. All methods for `print-object` must obey `*print-readably*`, which takes precedence over all other printer control variables. This includes both user-defined methods and implementation-defined methods.

If these rules are not obeyed, the results are undefined.

In general, the printer and the `print-object` methods should not rebind the print control variables as they operate recursively through the structure, but this is implementation-dependent.

In some implementations the stream argument passed to a `print-object` method is not the original stream but is an intermediate stream that implements part of the printer. Methods should therefore not depend on the identity of this stream.

All of the existing printing functions (`write`, `prin1`, `print`, `princ`, `pprint`, `write-to-string`, `prin1-to-string`, `princ-to-string`, the `~S` and `~A format` operations, and the `~B`, `~D`, `~E`, `~F`, `~G`, `~$`, `~O`, `~R`, and `~X format` operations when they encounter a non-numeric value) are required to be changed to go through the `print-object` generic function. Each implementation is required to replace its former implementation of printing with one or more `print-object` methods. Exactly which classes have methods for `print-object` is not specified; it would be valid for an implementation to have one default method that is inherited by all system-defined classes.

[Generic function] **reinitialize-instance** *instance* **&rest** *initargs*
[Primary method] **reinitialize-instance** (*instance* standard-object) **&rest** *initargs*

The generic function `reinitialize-instance` can be used to change the values of local slots according to initialization arguments. This generic function is called by the Meta-Object Protocol. It can also be called by users.

The system-supplied primary method for `reinitialize-instance` checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared valid. The method then calls the generic function `shared-initialize` with the following arguments: the instance, `nil` (which means no slots should be initialized according to their `:initform` forms) and the initialization arguments it received.

The *instance* argument is the object to be initialized.

The *initargs* argument consists of alternating initialization argument names and values.

The modified instance is returned as the result.

Initialization arguments are declared valid by using the `:initarg` option to `defclass`, or by defining methods for `reinitialize-instance` or `shared-initialize`. The keyword name of each keyword parameter specifier in the lambda-list of any method defined on `reinitialize-instance` or `shared-initialize` is declared a valid initialization argument name for all classes for which that method is applicable.

See sections 27.1.12, 27.1.9, 27.1.9 as well as `initialize-instance`, `slot-boundp`, `update-instance-for-redefined-class`, `update-instance-for-different-class`, `slot-makunbound`, and `shared-initialize`.

[Generic function] **remove-method** *generic-function method*
[Primary method] **remove-method**
(*generic-function* standard-generic-function) *method*

The generic function `remove-method` removes a method from a generic function. It destructively modifies the specified generic function and returns the modified generic function as its result.

The *generic-function* argument is a generic function object.

The *method* argument is a method object. The function `remove-method` does not signal an error if the method is not one of the methods on the generic function.

The modified generic function is returned. The result of `remove-method` is `eq` to the *generic-function* argument.

See `find-method`.

[Generic function] **shared-initialize** *instance slot-names* **&rest** *initargs*
[Primary method] **shared-initialize** (*instance* standard-object) *slot-names* **&rest** *initargs*

The generic function `shared-initialize` is used to fill the slots of an instance using initialization arguments and `:initform` forms. It is called when an instance is created, when an instance is re-initialized, when an instance is updated to conform to a redefined class, and when an instance is updated to conform to a different class. The generic function `shared-initialize` is called by the system-supplied primary method for `initialize-instance`, `reinitialize-instance`, `update-instance-for-redefined-class`, and `update-instance-for-different-class`.

The generic function `shared-initialize` takes the following arguments: the instance to be initialized, a specification of a set of names of slots accessible in that instance, and any number of initialization arguments. The arguments after the first two must form an initialization argument list. The system-supplied primary method on `shared-initialize` initializes the slots with values according to the initialization arguments and specified `:initform` forms. The second argument indicates which slots should be initialized according to their `:initform` forms if no initialization arguments are provided for those slots.

The system-supplied primary method behaves as follows, regardless of whether the slots are local or shared:

- If an initialization argument in the initialization argument list specifies a value for that slot, that value is stored into the slot, even if a value has already been stored in the slot before the method is run.
- Any slots indicated by the second argument that are still unbound at this point are initialized according to their `:initform` forms. For any such slot that has an `:initform` form, that form is evaluated in the lexical environment of its defining `defclass` form and the result is stored into the slot. For example, if a `:before` method stores a value in the slot, the `:initform` form will not be used to supply a value for the slot.
- The rules mentioned in section 27.1.9 are obeyed.

The *instance* argument is the object to be initialized.

The *slot-names* argument specifies the slots that are to be initialized according to their `:initform` forms if no initialization arguments apply. It is supplied in one of three forms as follows:

- It can be a list of slot names, which specifies the set of those slot names.
- It can be `nil`, which specifies the empty set of slot names.
- It can be the symbol `t`, which specifies the set of all of the slots.

The *initargs* argument consists of alternating initialization argument names and values.

The modified instance is returned as the result.

Initialization arguments are declared valid by using the `:initarg` option to `defclass`, or by defining methods for `shared-initialize`. The keyword name of each keyword parameter specifier in the lambda-list of any method defined on `shared-initialize` is declared a valid initialization argument name for all classes for which that method is applicable.

Implementations are permitted to optimize `:initform` forms that neither produce nor depend on side effects by evaluating these forms and storing them into slots before running any `initialize-instance` methods, rather than by handling them in the primary `initialize-instance` method. (This optimization might be implemented by having the `allocate-instance` method copy a prototype instance.)

Implementations are permitted to optimize default initial value forms for initialization arguments associated with slots by not actually creating the complete initialization argument list when the only method that would receive the complete list is the method on `standard-object`. In this case, default initial value forms can be treated like `:initform` forms. This optimization has no visible effects other than a performance improvement.

See sections 27.1.9, 27.1.9, 27.1.9 as well as `initialize-instance`, `reinitialize-instance`, `update-instance-for-redefined-class`, `update-instance-for-different-class`, `slot-boundp`, and `slot-makunbound`.

[Function] **slot-boundp** *instance slot-name*

The function `slot-boundp` tests whether a specific slot in an instance is bound.

The arguments are the instance and the name of the slot.

The function `slot-boundp` returns true or false.

This function allows for writing `:after` methods on `initialize-instance` in order to initialize only those slots that have not already been bound.

If no slot of the given name exists in the instance, `slot-missing` is called as follows:

```
(slot-missing (class-of instance)
              instance
              slot-name
              'slot-boundp)
```

The function `slot-boundp` is implemented using `slot-boundp-using-class`. See `slot-missing`.

[Function] **slot-exists-p** *object slot-name*

The function `slot-exists-p` tests whether the specified object has a slot of the given name.

The *object* argument is any object. The *slot-name* argument is a symbol.

The function `slot-exists-p` returns true or false.

The function `slot-exists-p` is implemented using `slot-exists-p-using-class`.

[Function] **slot-makunbound** *instance slot-name*

The function `slot-makunbound` restores a slot in an instance to the unbound state.

The arguments to `slot-makunbound` are the instance and the name of the slot.

The instance is returned as the result.

If no slot of the given name exists in the instance, `slot-missing` is called as follows:

```
(slot-missing (class-of instance)
              instance
              slot-name
              'slot-makunbound)
```

The function `slot-makunbound` is implemented using `slot-makunbound-using-class`. See `slot-missing`.

[Generic function] **slot-missing** *class object slot-name operation*
&optional *new-value*
[Primary method] **slot-missing** (*class* t) *object slot-name operation*
&optional *new-value*

The generic function `slot-missing` is invoked when an attempt is made to access a slot in an object whose metaclass is `standard-class` and the name of the slot provided is not a name of a slot in that class. The default method signals an error.

The generic function `slot-missing` is not intended to be called by programmers. Programmers may write methods for it.

The required arguments to `slot-missing` are the class of the object that is being accessed, the object, the slot name, and a symbol that indicates the operation that caused `slot-missing` to be invoked. The optional argument to `slot-missing` is used when the operation is attempting to set the value of the slot.

If a method written for `slot-missing` returns values, these values get returned as the values of the original function invocation.

The generic function `slot-missing` may be called during evaluation of `slot-value`, `(setf slot-value)`, `slot-boundp`, and `slot-makunbound`. For each of these operations the corresponding symbol for the *operation* argument is `slot-value`, `setf`, `slot-boundp`, and `slot-makunbound`, respectively.

The set of arguments (including the class of the instance) facilitates defining methods on the metaclass for `slot-missing`.

[Generic function] **slot-unbound** *class instance slot-name*
[Primary method] **slot-unbound** (*class* t) *instance slot-name*

The generic function `slot-unbound` is called when an unbound slot is read in an instance whose metaclass is `standard-class`. The default method signals an error.

The generic function `slot-unbound` is not intended to be called by programmers. Programmers may write methods for it. The function `slot-unbound` is called only by the function `slot-value-using-class` and thus indirectly by `slot-value`.

The arguments to `slot-unbound` are the class of the instance whose slot was accessed, the instance itself, and the name of the slot.

If a method written for `slot-unbound` returns values, these values get returned as the values of the original function invocation.

An unbound slot may occur if no `:initform` form was specified for the slot and the slot value has not been set, or if `slot-makunbound` has been called on the slot.

See `slot-makunbound`.

[Function] **slot-value** *object slot-name*

The function `slot-value` returns the value contained in the slot *slot-*

name of the given object. If there is no slot with that name, `slot-missing` is called. If the slot is unbound, `slot-unbound` is called.

The macro `setf` can be used with `slot-value` to change the value of a slot.

The arguments are the object and the name of the given slot.

The result is the value contained in the given slot.

If an attempt is made to read a slot and no slot of the given name exists in the instance, `slot-missing` is called as follows:

(slot-missing (class-of *instance*)
instance
slot-name
'slot-value)

If an attempt is made to write a slot and no slot of the given name exists in the instance, `slot-missing` is called as follows:

(slot-missing (class-of *instance*)
instance
slot-name
'setf
new-value)

The function `slot-value` is implemented using `slot-value-using-class`.

Implementations may optimize `slot-value` by compiling it in-line.

See `slot-missing` and `slot-unbound`.

[At this point the original CLOS report [5, 7] contained a specification for `symbol-macrolet`. This specification is omitted here. Instead, a description of `symbol-macrolet` appears with those of related constructs in chapter 7.—GLS]

[Generic function] **update-instance-for-different-class** *previous current* **&rest** *initargs*
[Primary method] **update-instance-for-different-class** (*previous* standard-object) (*current* standard-object) **&rest** *initargs*

The generic function `update-instance-for-different-class` is not intended to be called by programmers. Programmers may write methods for

it. This function is called only by the function `change-class`.

The system-supplied primary method on `update-instance-for-different-class` checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared valid. This method then initializes slots with values according to the initialization arguments and initializes the newly added slots with values according to their `:initform` forms. It does this by calling the generic function `shared-initialize` with the following arguments: the instance, a list of names of the newly added slots, and the initialization arguments it received. Newly added slots are those local slots for which no slot of the same name exists in the previous class.

Methods for `update-instance-for-different-class` can be defined to specify actions to be taken when an instance is updated. If only `:after` methods for `update-instance-for-different-class` are defined, they will be run after the system-supplied primary method for initialization and therefore will not interfere with the default behavior of `update-instance-for-different-class`.

The arguments to `update-instance-for-different-class` are computed by `change-class`. When `change-class` is invoked on an instance, a copy of that instance is made; `change-class` then destructively alters the original instance. The first argument to `update-instance-for-different-class`, *previous*, is that copy; it holds the old slot values temporarily. This argument has dynamic extent within `change-class`; if it is referenced in any way once `update-instance-for-different-class` returns, the results are undefined. The second argument to `update-instance-for-different-class`, *current*, is the altered original instance.

The intended use of *previous* is to extract old slot values by using `slot-value` or `with-slots` or by invoking a reader generic function, or to run other methods that were applicable to instances of the original class.

The *initargs* argument consists of alternating initialization argument names and values.

The value returned by `update-instance-for-different-class` is ignored by `change-class`.

See the example for the function `change-class`.

Initialization arguments are declared valid by using the `:initarg` option to `defclass`, or by defining methods for `update-instance-for-different-class` or `shared-initialize`. The

keyword name of each keyword parameter specifier in the lambda-list of any method defined on `update-instance-for-different-class` or `shared-initialize` is declared a valid initialization argument name for all classes for which that method is applicable.

Methods on `update-instance-for-different-class` can be defined to initialize slots differently from `change-class`. The default behavior of `change-class` is described in section 27.1.11.

See sections 27.1.11, 27.1.9, and 27.1.9 as well as `change-class` and `shared-initialize`.

[Generic function] **update-instance-for-redefined-class** *instance added-slots discarded-slots property-list* **&rest** *initargs*
[Primary method] **update-instance-for-redefined-class** (*instance* standard-object) *added-slots discarded-slots property-list* **&rest** *initargs*

The generic function `update-instance-for-redefined-class` is not intended to be called by programmers. Programmers may write methods for it. The generic function `update-instance-for-redefined-class` is called by the mechanism activated by `make-instances-obsolete`.

The system-supplied primary method on `update-instance-for-different-class` checks the validity of initialization arguments and signals an error if an initialization argument is supplied that is not declared valid. This method then initializes slots with values according to the initialization arguments and initializes the newly added slots with values according to their `:initform` forms. It does this by calling the generic function `shared-initialize` with the following arguments: the instance, a list of names of the newly added slots, and the initialization arguments it received. Newly added slots are those local slots for which no slot of the same name exists in the old version of the class.

When `make-instances-obsolete` is invoked or when a class has been redefined and an instance is being updated, a property list is created that captures the slot names and values of all the discarded slots with values in the original instance. The structure of the instance is transformed so that it conforms to the current class definition. The arguments to `update-instance-for-redefined-class` are this transformed instance, a list of the names of the new slots added to the instance, a list of the names of the old slots discarded from the instance, and the property list containing the slot names and values for slots that were discarded and had values. Included in this list of discarded slots are slots that were local in the old class and are shared in the new class.

The *initargs* argument consists of alternating initialization argument names and values.

The value returned by `update-instance-for-redefined-class` is ignored.

Initialization arguments are declared valid by using the `:initarg` option to `defclass` or by defining methods for `update-instance-for-redefined-class` or `shared-initialize`. The

keyword name of each keyword parameter specifier in the lambda-list of any method defined on `update-instance-for-redefined-class` or `shared-initialize` is declared a valid initialization argument name for all classes for which that method is applicable.

See sections 27.1.10, 27.1.9, and 27.1.9 as well as `shared-initialize` and `make-instances-obsolete`.

```
(defclass position () ())

(defclass x-y-position (position)
  ((x :initform 0 :accessor position-x)
   (y :initform 0 :accessor position-y)))

;;; It turns out polar coordinates are used more than Cartesian
;;; coordinates, so the representation is altered and some new
;;; accessor methods are added.

(defmethod update-instance-for-redefined-class :before
           ((pos x-y-position) added deleted plist &key)
  ;; Transform the x-y coordinates to polar coordinates
  ;; and store into the new slots.
  (let ((x (getf plist 'x))
        (y (getf plist 'y)))
    (setf (position-rho pos) (sqrt (+ (* x x) (* y y)))
          (position-theta pos) (atan y x))))

(defclass x-y-position (position)
    ((rho :initform 0 :accessor position-rho)
     (theta :initform 0 :accessor position-theta)))



;;; All instances of the old x-y-position class will be updated
;;; automatically.

;;; The new representation has the look and feel of the old one.
```

```
(defmethod position-x ((pos x-y-position))
  (with-slots (rho theta) pos (* rho (cos theta))))

(defmethod (setf position-x) (new-x (pos x-y-position))
  (with-slots (rho theta) pos
    (let ((y (position-y pos)))
      (setq rho (sqrt (+ (* new-x new-x) (* y y)))
            theta (atan y new-x))
      new-x)))

(defmethod position-y ((pos x-y-position))
  (with-slots (rho theta) pos (* rho (sin theta))))



(defmethod (setf position-y) (new-y (pos x-y-position))
  (with-slots (rho theta) pos
    (let ((x (position-x pos)))
      (setq rho (sqrt (+ (* x x) (* new-y new-y)))
            theta (atan new-y x))
      new-y)))
```

[Макрос] **with-accessors** ({slot-entry}*) instance-form {declaration}* {form}*

The macro `with-accessors` creates a lexical environment in which specified slots are lexically available through their accessors as if they were variables. The macro `with-accessors` invokes the appropriate accessors to access the specified slots. Both `setf` and `setq` can be used to set the value of the slot.

The result returned is that obtained by executing the forms specified by the *body* argument.

Example:

```
(with-accessors ((x position-x) (y position-y)) p1
  (setq x y))
```

A `with-accessors` expression of the form

(with-accessors ($slot_1$ ... $slot_n$) *instance*
 $declaration_1$ ... $declaration_m$)
 $form_1$ ... $form_k$)

expands into the equivalent of

(let ((*in instance*))
 (symbol-macrolet (($variable_1$ ($accessor_1$ *in*))
 ...
 ($variable_n$ ($accessor_n$ *in*)))
 $declaration_1$ ... $declaration_m$)
 $form_1$ ... $form_k$)

[X3J13 voted in March 1989 to modify the definition of `symbol-macrolet` substantially and also voted to allow declarations before the body of `symbol-macrolet` but with peculiar treatment of `special` and type declarations. The syntactic changes are reflected in this definition of `with-accessors`.—GLS]

See `with-slots` and `symbol-macrolet`.

[Специальный оператор] **with-added-methods** (function-name lambda-list
 [[↓option | {method-description}*]])
 {form}*

The `with-added-methods` special operator produces new generic functions and establishes new lexical function definition bindings. Each generic function is created by adding the set of methods specified by its method definitions to a copy of the lexically visible generic function of the same name and its methods. If such a generic function does not already exist, a new generic function is created; this generic function has lexical scope.

The special operator `with-added-methods` is used to define functions whose names are meaningful only locally and to execute a series of forms with these function definition bindings.

The names of functions defined by `with-added-methods` have lexical scope; they retain their local definitions only within the body of the `with-added-methods` construct. Any references within the body of the

`with-added-methods` construct to functions whose names are the same as those defined within the `with-added-methods` form are thus references to the local functions instead of to any global functions of the same names. The scope of these generic function definition bindings includes the method bodies themselves as well as the body of the `with-added-methods` construct.

The *function-name*, *option*, *method-qualifier*, and *specialized-lambda-list* arguments are the same as for `defgeneric`.

The body of each method is enclosed in an implicit block. If *function-name* is a symbol, this block bears the same name as the generic function. If *function-name* is a list of the form `(setf` *`symbol`*`)`, the name of the block is *symbol*.

The result returned by `with-added-methods` is the value or values of the last form executed. If no forms are specified, `with-added-methods` returns `nil`.

If a generic function with the given name already exists, the lambda-list specified in the `with-added-methods` form must be congruent with the lambda-lists of all existing methods on that function as well as with the lambda-lists of all methods defined by the `with-added-methods` form; otherwise an error is signaled.

If *function-name* specifies an existing generic function that has a different value for any of the following *option* arguments, the copy of that generic function is modified to have the new value: `:argument-precedence-order`, `declare`, `:documentation`, `:generic-function-class`, `:method-combination`.

If *function-name* specifies an existing generic function that has a different value for the `:method-class` *option* argument, that value is changed in the copy of that generic function, but any methods copied from the existing generic function are not changed.

If a function of the given name already exists, that function is copied into the default method for a generic function of the given name. Note that this behavior differs from that of `defgeneric`.

If a macro or special operator of the given name already exists, an error is signaled.

If there is no existing generic function, the *option* arguments have the same default values as the *option* arguments to `defgeneric`.

See `generic-labels`, `generic-flet`, `defmethod`, `defgeneric`, and `ensure-generic-function`.

[Макрос] **with-slots** ({slot-entry}*) instance-form {declaration}* {form}*

slot-entry ::= *slot-name* | `(variable-name slot-name)`

The macro `with-slots` creates a lexical context for referring to specified slots as though they were variables. Within such a context the value of the slot can be specified by using its slot name, as if it were a lexically bound variable. Both `setf` and `setq` can be used to set the value of the slot.

The macro `with-slots` translates an appearance of the slot name as a variable into a call to `slot-value`.

The result returned is that obtained by executing the forms specified by the *body* argument.

Example:

```
(with-slots (x y) position-1
  (sqrt (+ (* x x) (* y y))))

(with-slots ((x1 x) (y1 y)) position-1
  (with-slots ((x2 x) (y2 y)) position-2
    (psetf x1 x2
           y1 y2))))

(with-slots (x y) position
  (setq x (1+ x)
        y (1+ y)))
```

A `with-slots` expression of the form:

(with-slots (*slot-entry*$_1$ ... *slot-entry*$_n$) *instance*
 declaration$_1$ ... *declaration*$_m$)
 $form_1$ ... $form_k$)

expands into the equivalent of

(let ((*in instance*))
  (symbol-macrolet (Q_1 ... Q_n)
    $declaration_1$ ... $declaration_m$)
    $form_1$ ... $form_k$)

where Q_j is

(*slot-entry*$_j$ (slot-value *in* '*slot-entry*$_j$))

if *slot-entry*$_j$ is a symbol and is

(*variable-name*$_j$ (slot-value *in* '*slot-name*$_j$))

if *slot-entry*$_j$ is of the form (*variable-name*$_j$ *slot-name*$_j$).

[X3J13 voted in March 1989 to modify the definition of `symbol-macrolet` substantially and also voted to allow declarations before the body of `symbol-macrolet` but with peculiar treatment of `special` and type declarations. The syntactic changes are reflected in this definition of `with-slots`.—GLS]

See `with-accessors` and `symbol-macrolet`.

Таблица 27.1: Class Precedence Lists for Predefined Types

Predefined Common Lisp Type	Class Precedence List for Corresponding Class
array	(array t)
bit-vector	(bit-vector vector array sequence t)
character	(character t)
complex	(complex number t)
cons	(cons list sequence t)
float	(float number t)
function *	(function t)
hash-table *	(hash-table t)
integer	(integer rational number t)
list	(list sequence t)
null	(null symbol list sequence t)
number	(number t)
package *	(package t)
pathname *	(pathname t)
random-state *	(random-state t)
ratio	(ratio rational number t)
rational	(rational number t)
readtable *	(readtable t)
sequence	(sequence t)
stream *	(stream t)
string	(string vector array sequence t)
symbol	(symbol t)
t	(t)
vector	(vector array sequence t)

[An asterisk indicates a type added to this table as a consequence of a portion of the CLOS specification that was conditional on X3J13 voting to make that type disjoint from certain other built-in types .—GLS]

Глава 28

Conditions

Author: Kent M. Pitman preface: The language defined by the first edition contained an enormous lacuna: although facilities were specified for signaling errors, no means was defined for handling errors. This occurred not through neglect of the issue, but because this part of the Lisp language generally was in a state of flux. There were several proposals at the time. The committee, finding that it could not agree on any one proposal, agreed to disagree and omit error handling from Common Lisp for the time being. This defect has now been addressed.

X3J13 voted in June 1988 to adopt the Common Lisp Condition System as a part of the forthcoming draft Common Lisp standard. X3J13 voted in March 1989 to amend the specification of conditions to integrate them with the Common Lisp Object System (see chapter 27). X3J13 voted in June 1989 to amend the specification of restarts in certain ways. These amendments have been incorporated here with little further comment.

This chapter presents the bulk of the Common Lisp Condition System proposal, written by Kent M. Pitman and amended by X3J13. I have edited it only very lightly to conform to the overall style of this book and have inserted a small number of bracketed remarks identified by the initials GLS. Please see the Acknowledgments to this second edition for the author's acknowledgments to others who contributed to the Condition System proposal.

—Guy L. Steele Jr.

28.1 Introduction

Often we find it useful to describe a function in terms of its behavior in "normal situations." For example, we may say informally that the function `+` returns the sum of its arguments or that the function `read-char` returns the next available character on a given input stream.

Sometimes, however, an "exceptional situation" will arise that does not fit neatly into such descriptions. For example, `+` might receive an argument that is not a number, or `read-char` might receive as a single argument a stream that has no more available characters. This distinction between normal and exceptional situations is in some sense arbitrary but is often very useful in practice.

For example, suppose a function `f` were defined to allow only integer arguments but also guaranteed to detect and signal an error for non-integer arguments. Such a description is in fact internally inconsistent (that is, paradoxical) because the function's behavior is well-defined for non-integers. Yet we would not want this annoying paradox to force description of `f` as a function that accepts any kind of argument (just in case `f` is being called only as a quick way to signal an error, for example). Using the normal/exceptional distinction, we can say clearly that `f` accepts integers in the normal situation and signals an error in exceptional situations. Moreover, we can say that when we refer to the definition of a function informally, it is acceptable to speak only of its normal behavior. For example, we can speak informally about `f` as a function that accepts only integers without feeling that we are committing some awful fraud.

Not all exceptional situations are errors. For example, a program that is directing the typing of a long line of text may come to an end-of-line. It is possible that no real harm will result from failing to signal end-of-line to its caller because the operating system will simply force a carriage return on the output device, which will continue typing on the next line. However, it may still be interesting to establish a protocol whereby the printing program can inform its caller of end-of-line exceptions. The caller could then opt to deal with these situations in interesting ways at certain times. For example, a caller might choose to terminate printing, obtaining an end-of-line truncation. The important thing, however, is that the failure of the caller to provide advice about the situation need not prevent the printer program from operating correctly.

Mechanisms for dealing with exceptional situations vary widely. When

an exceptional situation is encountered, a program may attempt to handle it by returning a distinguished value, returning an additional value, setting a variable, calling a function, performing a special transfer of control, or stopping the program altogether and entering the debugger.

For the most part, the facilities described in this chapter do not introduce any fundamentally new way of dealing with exceptional situations. Rather, they encapsulate and formalize useful patterns of data and control flow that have been seen to be useful in dealing with exceptional situations.

A proper conceptual approach to errors should perhaps begin from first principles, with a discussion of *conditions* in general, and eventually work up to the concept of an *error* as just one of the many kinds of conditions. However, given the primitive state of error-handling technology, a proper buildup may be as inappropriate as requiring that a beggar learn to cook a gourmet meal before being allowed to eat. Thus, we deal first with the essentials—error handling—and then go back later to fill in the missing details.

28.2 Changes in Terminology

In this section, we introduce changes to the terminology defined in section 1.2.4.

A *condition* is an interesting situation in a program that has been detected and announced. Later we allow this term also to refer to objects that programs use to represent such situations.

An *error* is a condition in which normal program execution may not continue without some form of intervention (either interactively by the user or under some sort of program control, as described below).

The process by which a condition is formally announced by a program is called *signaling*. The function `signal` is the primitive mechanism by which such announcement is done. Other abstractions, such as `error` and `cerror`, are built using `signal`.

The first edition is ambiguous about the reason why a particular program action "is an error." There are two principal reasons why an action may be an error without being required to signal an error:

- Detecting the error might be prohibitively expensive.

  For example, `(+ nil 3)` is an error. It is likely that the designers of Common Lisp believed this would be an error in all implementations

but felt it might be excessively expensive to detect the problem in compiled code on stock hardware, so they did not require that it signal an error.

- Some implementations might implement the behavior as an extension.

  For example, `(loop for x from 1 to 3 do (print x))` is an error because `loop` is not defined to take atoms in its body. In fact, however, some implementations offer an extension that makes this well-defined. In order to leave room for such extensions, the first edition used the "is an error" terminology to keep implementors from being forced to signal an error in the extended implementations.

  [This example was written well before the vote by X3J13 in January 1989 to add exactly this extension to the forthcoming draft standard (see chapter 25).—GLS]

In this chapter, we use the following terminology. [Compare this to the terminology presented in section 27.1.1.—GLS]

- If the signaling of a condition or error is part of a function's contract in all situations, we say that it "signals" or "must signal" that condition or error.

- If the signaling of a condition or error is optional for some important reason (such as performance), we say that the program "might signal" that condition or error. In this case, we are defining the operation to be illegal in all implementations, but allowing some implementations to fail to detect the error.

- If an action is left undefined for the sake of implementation-dependent extension, we say that it "is undefined" or "has undefined effect." This means that it is not possible to depend portably upon the effects of that action. A program that has undefined effect may enter the debugger, transfer control, or modify data in unpredictable ways.

- In the special case where only the return value of an operation is not well defined but any side effect and transfer-of-control behavior is well defined, we say that it has "undefined value." In this case, the number and nature of the return values is not defined, but the function can reasonably be expected to return. It is worth noting that under this

description, there are some (though not many) legitimate ways in which such return value(s) can be used. For example, if the function `foo` has no side effects and undefined value, the expression `(length (list (foo)))` is completely well defined even for portable code. However, the effect of `(print (list (foo)))` is not well defined.

28.3 Survey of Concepts

This section discusses various aspects of the condition system by topic, illustrating them with extensive examples. The next section contains definitions of specific functions, macros, and other facilities.

28.3.1 Signaling Errors

Conceptually, signaling an error in a program is an admission by that program that it does not know how to continue and requires external intervention. Once an error is signaled, any decision about how to continue must come from the "outside."

The simplest way to signal an error is to use the `error` function with `format`-style arguments describing the error for the sake of the user interface. If `error` is called and there are no active handlers (described in sections 28.3.2 and 28.3.3), the debugger will be entered and the error message will be typed out. For example:

```
Lisp> (defun factorial (x)
       (cond ((or (not (typep x 'integer)) (minusp x))
              (error "~S is not a valid argument to FACTORIAL."
                     x))
             ((zerop x) 1)
             (t (* x (factorial (- x 1))))))
 ⇒ FACTORIAL
Lisp> (factorial 20)
 ⇒ 2432902008176640000
Lisp> (factorial -1)
Error: -1 is not a valid argument to FACTORIAL.
To continue, type :CONTINUE followed by an option number:
 1: Return to Lisp Toplevel.
Debug>
```

In general, a call to `error` cannot directly return. Unless special work has been done to override this behavior, the debugger will be entered and there will be no option to simply continue.

The only exception may be that some implementations may provide debugger commands for interactively returning from individual stack frames; even then, however, such commands should never be used except by someone who has read the erring code and understands the consequences of continuing from that point. In particular, the programmer should feel confident about writing code like this:

```
(defun wargames:no-win-scenario ()
  (when (true) (error "Pushing the button would be stupid."))
  (push-the-button))
```

In this scenario, there should be no chance that the function `error` will return and the button will be pushed.

Примечание: It should be noted that the notion of "no chance" that the button will be pushed is relative only to the language model; it assumes that the language is accurately implemented. In practice, compilers have bugs, computers have glitches, and users have been known to interrupt at inopportune moments and use the debugger to return from arbitrary stack frames. Such violations of the language model are beyond the scope of the condition system but not necessarily beyond the scope of potential failures that the programmer should consider and defend against. The possibility of such unusual failures may of course also influence the design of code meant to handle less drastic situations, such as maintaining a database uncorrupted.—KMP and GLS

In some cases, the programmer may have a single, well-defined idea of a reasonable recovery strategy for this particular error. In that case, he can use the function `cerror`, which specifies information about what would happen if the user did simply continue from the call to `cerror`. For example:

```
Lisp> (defun factorial (x)
        (cond ((not (typep x 'integer))
               (error "~S is not a valid argument to FACTORIAL."
                      x))
```

```
          ((minusp x)
           (let ((x-magnitude (- x)))
             (cerror "Compute -(~D!) instead."
                     "(-~D)! is not defined." x-magnitude)
             (- (factorial x-magnitude))))
          ((zerop x) 1)
          (t (* x (factorial (- x 1))))))
⇒ FACTORIAL
Lisp> (factorial -3)
Error: (-3)! is not defined.
To continue, type :CONTINUE followed by an option number:
 1: Compute -(3!) instead.
 2: Return to Lisp Toplevel.
Debug> :continue 1
⇒ -6
```

28.3.2 Trapping Errors

By default, a call to `error` will force entry into the debugger. You can override that behavior in a variety of ways. The simplest (and most blunt) tool for inhibiting entry to the debugger on an error is to use `ignore-errors`. In the normal situation, forms in the body of `ignore-errors` are evaluated sequentially and the last value is returned. If a condition of type `error` is signaled, `ignore-errors` immediately returns two values, namely `nil` and the condition that was signaled; the debugger is not entered and no error message is printed. For example:

```
Lisp> (setq filename "nosuchfile")
⇒ "nosuchfile"
Lisp> (ignore-errors (open filename :direction :input))
⇒ NIL and #<FILE-ERROR 3437523>
```

The second return value is an object that represents the kind of error. This is explained in greater detail in section 28.3.4.

In many cases, however, `ignore-errors` is not desirable because it deals with too many kinds of errors. Contrary to the belief of some, a program that does not enter the debugger is not necessarily better than one that does. Excessive use of `ignore-errors` may keep the program out of the debugger, but it may not increase the program's reliability, because the program may continue to run after encountering errors other than those you meant to work past. In general, it is better to attempt to deal only with the particular kinds of errors that you believe could legitimately happen. That way, if an unexpected error comes along, you will still find out about it.

`ignore-errors` is a useful special case built from a more general facility, `handler-case`, that allows the programmer to deal with particular kinds of conditions (including non-error conditions) without affecting what happens when other kinds of conditions are signaled. For example, an effect equivalent to that of `ignore-errors` above is achieved in the following example:

```
Lisp> (setq filename "nosuchfile")
 ⇒ "nosuchfile"
Lisp> (handler-case (open filename :direction :input)
        (error (condition)
          (values nil condition)))
 ⇒ NIL and #<FILE-ERROR 3437525>
```

However, using `handler-case`, one can indicate a more specific condition type than just "error." Condition types are explained in detail later, but the syntax looks roughly like the following:

```
Lisp> (makunbound 'filename)
 ⇒ FILENAME
Lisp> (handler-case (open filename :direction :input)
        (file-error (condition)
          (values nil condition)))
Error: The variable FILENAME is unbound.
To continue, type :CONTINUE followed by an option number:
 1: Retry getting the value of FILENAME.
 2: Specify a value of FILENAME to use this time.
 3: Specify a value of FILENAME to store and use.
 4: Return to Lisp Toplevel.
Debug>
```

28.3.3 Handling Conditions

Blind transfer of control to a `handler-case` is only one possible kind of recovery action that can be taken when a condition is signaled. The low-level mechanism offers great flexibility in how to continue once a condition has been signaled.

The basic idea behind condition handling is that a piece of code called the *signaler* recognizes and announces the existence of an exceptional situation using `signal` or some function built on `signal` (such as `error`).

The process of signaling involves the search for and invocation of a *handler*, a piece of code that will attempt to deal appropriately with the situation.

If a handler is found, it may either *handle* the situation, by performing some non-local transfer of control, or *decline* to handle it, by failing to perform a non-local transfer of control. If it declines, other handlers are sought.

Since the lexical environment of the signaler might not be available to handlers, a data structure called a *condition* is created to represent explicitly the relevant state of the situation. A condition either is created explicitly using `make-condition` and then passed to a function such as `signal`, or is created implicitly by a function such as `signal` when given appropriate non-condition arguments.

In order to handle the error, a handler is permitted to use any non-local transfer of control such as `go` to a tag in a `tagbody`, `return` from a `block`, or `throw` to a `catch`. In addition, structured abstractions of these primitives are provided for convenience in exception handling.

A handler can be made dynamically accessible to a program by use of `handler-bind`. For example, to create a handler for a condition of type `arithmetic-error`, one might write:

(handler-bind ((arithmetic-error *handler*))*body*)

The handler is a function of one argument, the condition. If a condition of the designated type is signaled while the *body* is executing (and there are no intervening handlers), the handler would be invoked on the given condition, allowing it the option of transferring control. For example, one might write a macro that executes a body, returning either its value(s) or the two values `nil` and the condition:

```
(defmacro without-arithmetic-errors (&body forms)
  (let ((tag (gensym)))
    `(block ,tag
       (handler-bind ((arithmetic-error
                       #'(lambda (c)     ;Argument c is a condition
                           (return-from ,tag (values nil c)))))
         ,@body))))
```

The handler is executed in the dynamic context of the signaler, except that the set of available condition handlers will have been rebound to the value that was active at the time the condition handler was made active. If a handler declines (that is, it does not transfer control), other handlers are sought. If no handler is found and the condition was signaled by `error` or `cerror` (or some function such as `assert` that behaves like these functions), the debugger is entered, still in the dynamic context of the signaler.

28.3.4 Object-Oriented Basis of Condition Handling

Of course, the ability of the handler to usefully handle an exceptional situation is related to the quality of the information it is provided. For example, if all errors were signaled by

(error "*some format string*")

then the only piece of information that would be accessible to the handler would be an object of type `simple-error` that had a slot containing the format string.

If this were done, `string-equal` would be the preferred way to tell one error from another, and it would be very hard to allow flexibility in the presentation of error messages because existing handlers would tend to be broken by even tiny variations in the wording of an error message. This phenomenon has been the major failing of most error systems previously available in Lisp. It is fundamentally important to decouple the error message string (the human interface) from the objects that formally represent the error state (the program interface). We therefore have the notion of typed

conditions, and of formal operations on those conditions that make them inspectable in a structured way.

This object-oriented approach to condition handling has the following important advantages over a text-based approach:

- Conditions are classified according to subtype relationships, making it easy to test for categories of conditions.
- Conditions have named slot values through which parameters are conveyed from the program that signals the condition to the program that handles it.
- Inheritance of methods and slots reduces the amount of explicit specification necessary to achieve various interesting effects.

Some condition types are defined by this document, but the set of condition types is extensible using `define-condition`. Common Lisp condition types are in fact CLOS classes, and condition objects are ordinary CLOS objects; `define-condition` merely provides an abstract interface that is a bit more convenient than `defclass` for defining conditions.

Here, as an example, we define a two-argument function called `divide` that is patterned after the `/` function but does some stylized error checking:

```
(defun divide (numerator denominator)
  (cond ((or (not (numberp numerator))
             (not (numberp denominator)))
         (error "(DIVIDE '~S '~S) - Bad arguments."
                numerator denominator))
        ((zerop denominator)
         (error 'division-by-zero
                :operator 'divide
                :operands (list numerator denominator)))
        (t ...)))
```

Note that in the first clause we have used `error` with a string argument and in the second clause we have named a particular condition type, `division-by-zero`. In the case of a string argument, the condition type that will be signaled is `simple-error`.

The particular kind of error that is signaled may be important in cases where handlers are active. For example, `simple-error` inherits from type `error`, which in turn inherits from type `condition`. On the other hand, `division-by-zero` inherits from `arithmetic-error`, which inherits from `error`, which inherits from `condition`. So if a handler existed for `arithmetic-error` while a `division-by-zero` condition was signaled, that handler would be tried; however, if a `simple-error` condition were signaled in the same context, the handler for type `arithmetic-error` would not be tried.

28.3.5 Restarts

In older Lisp dialects (such as MacLisp), an attempt to signal an error of a given type often carried with it an implicit promise to support the standard recovery strategy for that type of error. If the signaler knew the type of error but for whatever reason was unable to deal with the standard recovery strategy for that kind of error, it was necessary to signal an untyped error (for which there was no defined recovery strategy). This sometimes led to confusion when people signaled typed errors without realizing the full implications of having done so, but more often than not it meant that users simply avoided typed errors altogether.

The Common Lisp Condition System, which is modeled after the Zetalisp condition system, corrects this troublesome aspect of previous Lisp dialects by creating a clear separation between the act of signaling an error of a particular type and the act of saying that a particular way of recovery is appropriate. In the `divide` example above, simply signaling an error does not imply a willingness on the part of the signaler to cooperate in any corrective action. For example, the following sample interaction illustrates that the only recovery action offered for this error is "Return to Lisp Toplevel":

```
Lisp> (+ (divide 3 0) 7)
Error: Attempt to divide 3 by 0.
To continue, type :CONTINUE followed by an option number:
 1: Return to Lisp Toplevel.
Debug> :continue 1
Returned to Lisp Toplevel.
Lisp>
```

When an error is detected and the function `error` is called, execution cannot continue normally because `error` will not directly return. Control can be transferred to other points in the program, however, by means of specially established "restarts."

28.3.6 Anonymous Restarts

The simplest kind of restart involves structured transfer of control using a macro called `restart-case`. The `restart-case` form allows execution of a piece of code in a context where zero or more restarts are active, and where if one of those restarts is "invoked," control will be transferred to the corresponding clause in the `restart-case` form. For example, we could rewrite the previous `divide` example as follows.

```
(defun divide (numerator denominator)
  (loop
    (restart-case
        (return
          (cond ((or (not (numberp numerator))
                     (not (numberp denominator)))
                 (error "(DIVIDE '~S '~S) - Bad arguments."
                        numerator denominator))
                ((zerop denominator)
                 (error 'division-by-zero
                        :operator 'divide
                        :operands (list numerator denominator)))
                (t ...)))
      (nil (arg1 arg2)
        :report "Provide new arguments for use by DIVIDE."
        :interactive
          (lambda ()
            (list (prompt-for 'number "Numerator: ")
                  (prompt-for 'number "Denominator: ")))
        (setq numerator arg1 denominator arg2))
      (nil (result)
        :report "Provide a value to return from DIVIDE."
        :interactive
```

```
    (lambda () (list (prompt-for 'number "Result: ")))
  (return result)))))
```

Примечание: The function `prompt-for` used in this chapter in a number of places is not a part of Common Lisp. It is used in the examples in this chapter only to keep the presentation simple. It is assumed to accept a type specifier and optionally a format string and associated arguments. It uses the format string and associated arguments as part of an interactive prompt, and uses `read` to read a Lisp object; however, only an object of the type indicated by the type specifier is accepted.

The question of whether or not `prompt-for` (or something like it) would be a useful addition to Common Lisp is under consideration by X3J13, but as of January 1989 no action has been taken. In spite of its use in a number of examples, nothing in the Common Lisp Condition System depends on this function.

In the example, the `nil` at the head of each clause means that it is an "anonymous" restart. Anonymous restarts are typically invoked only from within the debugger. As we shall see later, it is possible to have "named restarts" that may be invoked from code without the need for user intervention.

If the arguments to anonymous restarts are not optional, then special information must be provided about what the debugger should use as arguments. Here the `:interactive` keyword is used to specify that information.

The `:report` keyword introduces information to be used when presenting the restart option to the user (by the debugger, for example).

Here is a sample interaction that takes advantage of the restarts provided by the revised definition of `divide`:

```
Lisp> (+ (divide 3 0) 7)
Error: Attempt to divide 3 by 0.
To continue, type :CONTINUE followed by an option number:
 1: Provide new arguments for use by the DIVIDE function.
 2: Provide a value to return from the DIVIDE function.
 3: Return to Lisp Toplevel.
Debug> :continue 1
1
Numerator: 4
Denominator: 2
 ⇒ 9
```

28.3.7 Named Restarts

In addition to anonymous restarts, one can have named restarts, which can be invoked by name from within code. As a trivial example, one could write

```
(restart-case (invoke-restart 'foo 3)
  (foo (x) (+ x 1)))
```

to add `3` to `1`, returning `4`. This trivial example is conceptually analogous to writing:

```
(+ (catch 'something (throw 'something 3)) 1)
```

For a more realistic example, the code for the function `symbol-value` might signal an unbound variable error as follows:

```
(restart-case (error "The variable ~S is unbound." variable)
  (continue ()
     :report
       (lambda (s)     ;Argument s is a stream
         (format s "Retry getting the value of ~S." variable))
    (symbol-value variable))
  (use-value (value)
     :report
       (lambda (s)     ;Argument s is a stream
         (format s "Specify a value of ~S to use this time."
                 variable))
    value)
  (store-value (value)
     :report
       (lambda (s)     ;Argument s is a stream
         (format s "Specify a value of ~S to store and use."
                 variable))
    (setf (symbol-value variable) value)
    value))
```

If this were part of the implementation of `symbol-value`, then it would be possible for users to write a variety of automatic handlers for unbound variable errors. For example, to make unbound variables evaluate to themselves, one might write

```
(handler-bind ((unbound-variable
                #'(lambda (c)     ;Argument c is a condition
                    (when (find-restart 'use-value)
                      (invoke-restart 'use-value
                                      (cell-error-name c))))))
  body)
```

28.3.8 Restart Functions

For commonly used restarts, it is conventional to define a program interface that hides the use of `invoke-restart`. Such program interfaces to restarts are called *restart functions*.

The normal convention is for the function to share the name of the restart. The pre-defined functions `abort`, `continue`, `muffle-warning`, `store-value`, and `use-value` are restart functions. With `use-value` the above example of `handler-bind` could have been written more concisely as

```
(handler-bind ((unbound-variable
                #'(lambda (c)     ;Argument c is a condition
                    (use-value (cell-error-name c)))))
  body)
```

28.3.9 Comparison of Restarts and Catch/Throw

One important feature that `restart-case` (or `restart-bind`) offers that `catch` does not is the ability to reason about the available points to which control might be transferred without actually attempting the transfer. One could, for example, write

```
(ignore-errors (throw ...))
```

which is a sort of poor man's variation of

```
(when (find-restart 'something)
  (invoke-restart 'something))
```

but there is no way to use `ignore-errors` and `throw` to simulate something like

```
(when (and (find-restart 'something)
           (find-restart 'something-else))
  (invoke-restart 'something))
```

or even just

```
(when (and (find-restart 'something)
           (yes-or-no-p "Do something? "))
  (invoke-restart 'something))
```

because the degree of inspectability that comes with simply writing

```
(ignore-errors (throw ...))
```

is too primitive—getting the desired information also forces transfer of control, perhaps at a time when it is not desirable.

Many programmers have previously evolved strategies like the following on a case-by-case basis:

```
(defvar *foo-tag-is-available* nil)

(defun fn-1 ()
  (catch 'foo
    (let ((*foo-tag-is-available* t))
      ... (fn-2) ...)))

(defun fn-2 ()
  ...
  (if *foo-tag-is-available* (throw 'foo t))
  ...)
```

The facility provided by `restart-case` and `find-restart` is intended to provide a standardized protocol for this sort of information to be communicated between programs that were developed independently so that individual variations from program to program do not thwart the overall modularity and debuggability of programs.

Another difference between the restart facility and the `catch`/`throw` facility is that a `catch` with any given tag completely shadows any outer pending `catch` that uses the same tag. Because of the presence of `compute-restarts`, however, it is possible to see shadowed restarts, which may be very useful in some situations (particularly in an interactive debugger).

28.3.10 Generalized Restarts

`restart-case` is a mechanism that allows only imperative transfer of control for its associated restarts. `restart-case` is built on a lower-level mechanism called `restart-bind`, which does not force transfer of control.

`restart-bind` is to `restart-case` as `handler-bind` is to `handler-case`. The syntax is

(restart-bind ((*name function* . *options*)) . *body*)

The *body* is executed in a dynamic context within which the *function* will be called whenever `(invoke-restart 'name)` is executed. The *options* are keyword-style and are used to pass information such as that provided with the `:report` keyword in `restart-case`.

A `restart-case` expands into a call to `restart-bind` where the function simply does an unconditional transfer of control to a particular body of code, passing along "argument" information in a structured way.

It is also possible to write restarts that do not transfer control. Such restarts may be useful in implementing various special commands for the debugger that are of interest only in certain situations. For example, one might imagine a situation where file space was exhausted and the following was done in an attempt to free space in directory `dir`:

```
(restart-bind ((nil #'(lambda () (expunge-directory dir))
                    :report-function
                      #'(lambda (stream)
                          (format stream "Expunge ~A."
                                  (directory-namestring dir)))))
  (cerror "Try this file operation again."
          'directory-full :directory dir))
```

In this case, the debugger might be entered and the user could first perform the expunge (which would not transfer control from the debugger context) and then retry the file operation:

```
Lisp> (open "FOO" :direction :output)
Error: The directory PS:<JDOE> is full.
To continue, type :CONTINUE followed by an option number:
 1: Try this file operation again.
 2: Expunge PS:<JDOE>.
 3: Return to Lisp Toplevel.
Debug> :continue 2
Expunging PS:<JDOE> ... 3 records freed.
Debug> :continue 1
 ⇒ #<OUTPUT-STREAM "PS:<JDOE>FOO.LSP" 2323473>
```

28.3.11 Interactive Condition Handling

When a program does not know how to continue, and no active handler is able to advise it, the "interactive condition handler," or "debugger," can be entered. This happens implicitly through the use of functions such as `error` and `cerror`, or explicitly through the use of the function `invoke-debugger`.

The interactive condition handler never returns directly; it returns only through structured non-local transfer of control to specially defined restart points that can be set up either by the system or by user code. The mechanisms that support the establishment of such structured restart points for portable code are outlined in sections 28.3.5 through 28.3.10.

Actually, implementations may also provide extended debugging facilities that allow return from arbitrary stack frames. Although such commands are frequently useful in practice, their effects are implementation-dependent because they violate the Common Lisp program abstraction. The effect of using such commands is undefined with respect to Common Lisp.

28.3.12 Serious Conditions

The `ignore-errors` macro will trap conditions of type `error`. There are, however, conditions that are not of type `error`.

Some conditions are not considered errors but are still very serious, so we call them *serious conditions* and we use the type `serious-condition` to

represent them. Conditions such as those that might be signaled for "stack overflow" or "storage exhausted" are in this category.

The type `error` is a subtype of `serious-condition`, and it would technically be correct to use the term "serious condition" to refer to all serious conditions whether errors or not. However, normally we use the term "serious condition" to refer to things of type `serious-condition` but not of type `error`.

The point of the distinction between errors and other serious conditions is that some conditions are known to occur for reasons that are beyond the scope of Common Lisp to specify clearly. For example, we know that a stack will generally be used to implement function calling, and we know that stacks tend to be of finite size and are prone to overflow. Since the available stack size may vary from implementation to implementation, from session to session, or from function call to function call, it would be confusing to have expressions such as `(ignore-errors (+ a b))` return a number sometimes and `nil` other times if `a` and `b` were always bound to numbers and the stack just happened to overflow on a particular call. For this reason, only conditions of type `error` and not all conditions of type `serious-condition` are trapped by `ignore-errors`. To trap other conditions, a lower-level facility must be used (such as `handler-bind` or `handler-case`).

By convention, the function `error` is preferred over `signal` to signal conditions of type `serious-condition` (including those of type `error`). It is the use of the function `error`, and not the type of the condition being signaled, that actually causes the debugger to be entered.

Несовместимость: The Common Lisp Condition System differs from that of Zetalisp in this respect. In Zetalisp the debugger is entered for an unhandled signal if the `error` function is used *or* if the condition is of type `error`.

28.3.13 Non-Serious Conditions

Some conditions are neither errors nor serious conditions. They are signaled to give other programs a chance to intervene, but if no action is taken, computation simply continues normally.

For example, an implementation might choose to signal a non-serious (and implementation-dependent) condition called `end-of-line` when output reaches the last character position on a line of character output. In such an implementation, the signaling of this condition might allow a convenient way

for other programs to intervene, producing output that is truncated at the end of a line.

By convention, the function `signal` is used to signal conditions that are not serious. It would be possible to signal serious conditions using `signal`, and the debugger would not be entered if the condition went unhandled. However, by convention, handlers will generally tend to assume that serious conditions and errors were signaled by calling the `error` function (and will therefore force entry to the interactive condition handler) and that they should work to avoid this.

28.3.14 Condition Types

Some types of conditions are predefined by the system. All types of conditions are subtypes of `condition`. That is, `(typep` *x* `'condition)` is true if and only if the value of x is a condition.

Implementations supporting multiple (or non-hierarchical) type inheritance are expressly permitted to exploit multiple inheritance in the tree of condition types as implementation-dependent extensions, as long as such extensions are compatible with the specifications in this chapter. [X3J13 voted in March 1989 to integrate the Condition System and the Object System, so multiple inheritance is always available for condition types.—GLS]

In order to avoid problems in portable code that runs both in systems with multiple type inheritance and in systems without it, programmers are explicitly warned that while all correct Common Lisp implementations will ensure that `(typep c 'condition)` is true for all conditions c (and all subtype relationships indicated in this chapter will also be true), it should *not* be assumed that two condition types specified to be subtypes of the same third type are disjoint. (In some cases, disjoint subtypes are identified explicitly, but such disjointness is not to be assumed by default.) For example, it follows from the subtype descriptions contained in this chapter that in all implementations `(typep c 'control-error)` implies `(typep c 'error)`, but note that `(typep c 'control-error)` does *not* imply `(not (typep c 'cell-error))`.

28.3.15 Signaling Conditions

When a condition is signaled, the system tries to locate the most appropriate handler for the condition and to invoke that handler.

Handlers are established dynamically using `handler-bind` or abstractions built on `handler-bind`.

If an appropriate handler is found, it is called. In some circumstances, the handler may *decline* simply by returning without performing a non-local transfer of control. In such cases, the search for an appropriate handler is picked up where it left off, as if the called handler had never been present.

If no handler is found, or if all handlers that were found decline, `signal` returns `nil`.

Although it follows from the description above, it is perhaps worth noting explicitly that the lookup procedure described here will prefer a general but more (dynamically) local handler over a specific but less (dynamically) local handler. Experience with existing condition systems suggests that this is a reasonable approach and works adequately in most situations. Some care should be taken when binding handlers for very general kinds of conditions, such as is done in `ignore-errors`. Often, binding for a more specific condition type than `error` is more appropriate.

28.3.16 Resignaling Conditions

[The contents of this section are still a subject of some debate within X3J13. The reader may wish to take this section with a grain of salt.—GLS]

Note that signaling a condition has no side effect on that condition, and that there is no dynamic state contained in a condition object. As such, it may at times be reasonable and appropriate to consider caching condition objects for repeated use, re-signaling conditions from within handlers, or saving conditions away somewhere and re-signaling them later.

For example, it may be desirable for the system to pre-allocate objects of type `storage-condition` so that they can be signaled when needed without attempting to allocate more storage.

28.3.17 Condition Handlers

A *handler* is a function of one argument, the condition to be handled. The handler may inspect the object to be sure it is "interested" in handling the condition.

A handler is executed in the dynamic context of the signaler, except that the set of available condition handlers will have been rebound to the value that was active at the time the condition handler was made active. The intent of this is to prevent infinite recursion because of errors in a condition handler.

After inspecting the condition, the handler should take one of the following actions:

- It might *decline* to handle the condition (by simply returning). When this happens, the returned values are ignored and the effect is the same as if the handler had been invisible to the mechanism seeking to find a handler. The next handler in line will be tried, or if no such handler exists, the condition will go unhandled.
- It might *handle* the condition (by performing some non-local transfer of control). This may be done either primitively using `go`, `return`, or `throw`, or more abstractly using a function such as `abort` or `invoke-restart`.
- It might signal another condition.
- It might invoke the interactive debugger.

In fact, the latter two actions (signaling another condition or entering the debugger) are really just ways of putting off the decision to either handle or decline, or trying to get someone else to make such a decision. Ultimately, all a handler can do is to handle or decline to handle.

28.3.18 Printing Conditions

When `*print-escape*` is `nil` (for example, when the `princ` function or the `~A` directive is used with `format`), the report method for the condition will be invoked. This will be done automatically by functions such as `invoke-debugger`, `break`, and `warn`, but there may still be situations in

which it is desirable to have a condition report under explicit user control. For example,

```
(let ((form '(open "nosuchfile")))
  (handler-case (eval form)
    (serious-condition (c)
      (format t "~&Evaluation of ~S failed:~%~A" form c))))
```

might print something like

```
Evaluation of (OPEN "nosuchfile") failed:
The file "nosuchfile" was not found.
```

Some suggestions about the form of text typed by report methods:

- The message should generally be a complete sentence, beginning with a capital letter and ending with appropriate punctuation (usually a period).
- The message should *not* include any introductory text such as "`Error:`" or "`Warning:`" and should not be followed by a trailing newline. Such text will be added as may be appropriate to context by the routine invoking the report method.
- Except where unavoidable, the tab character (which is only semi-standard anyway) should not be used in error messages. Its effect may vary from one implementation to another and may cause problems even within an implementation because it may do different things depending on the column at which the error report begins.
- Single-line messages are preferred, but newlines in the middle of long messages are acceptable.
- If any program (for example, the debugger) displays messages indented from the prevailing left margin (for example, indented seven spaces because they are prefixed by the seven-character herald "`Error: `"), then that program will take care of inserting the appropriate indentation into the extra lines of a multi-line error message. Similarly, a program

that prefixes error messages with semicolons so that they appear to be comments should take care of inserting a semicolon at the beginning of each line in a multi-line error message. (These rules are important because, even within a single implementation, there may be more than one program that presents error messages to the user, and they may use different styles of presentation. The caller of `error` cannot anticipate all such possible styles, and so it is incumbent upon the presenter of the message to make any necessary adjustments.)

[Note: These recommendations expand upon those in section **??**.—GLS]

When `*print-escape*` is not `nil`, the object should print in some useful (but usually fairly abbreviated) fashion according to the style of the implementation. It is not expected that a condition will be printed in a form suitable for `read`. Something like `#<ARITHMETIC-ERROR 1734>` is fine.

X3J13 voted in March 1989 to integrate the Condition System and the Object System. In the original Condition System proposal, no function was provided for directly accessing or setting the printer for a condition type, or for invoking it; the techniques described above were the sole interface to reporting. The vote specified that, in CLOS terms, condition reporting is mediated through the `print-object` method for the condition type (that is, class) in question, with `*print-escape*` bound to `nil`. Specifying `(:report` *fn*`)` to `define-condition` when defining condition type *C* is equivalent to a separate method definition:

```
(defmethod print-object ((x C) stream)
  (if *print-escape*
      (call-next-method)
      (funcall #'fn x stream)))
```

Note that the method uses *fn* to print the condition only when `*print-escape*` has the value `nil`.

28.4 Program Interface to the Condition System

This section describes functions, macros, variables, and condition types associated with the Common Lisp Condition System.

28.4.1 Signaling Conditions

The functions in this section provide various mechanisms for signaling warnings, breaks, continuable errors, and fatal errors.

[Function] **error** *datum* **&rest** *arguments*

[This supersedes the description of `error` given in section **??**.—GLS]

Invokes the signal facility on a condition. If the condition is not handled, `(invoke-debugger` *`condition`*`)` is executed. As a consequence of calling `invoke-debugger`, `error` never directly returns to its caller; the only exit from this function can come by non-local transfer of control in a handler or by use of an interactive debugging command.

If *datum* is a condition, then that condition is used directly. In this case, it is an error for the list of *arguments* to be non-empty; that is, `error` must have been called with exactly one argument, the condition.

If *datum* is a condition type (a class or class name), then the condition used is effectively the result of `(apply #'make-condition` *`datum arguments`*`)`.

If *datum* is a string, then the condition used is effectively the result of

```
(make-condition 'simple-error
                :format-string datum
                :format-arguments arguments)
```

[Function] **cerror** *continue-format-string datum* **&rest** *arguments*

[This supersedes the description of `cerror` given in section **??**.—GLS]

The function `cerror` invokes the error facility on a condition. If the condition is not handled, `(invoke-debugger` *`condition`*`)` is executed. While signaling is going on, and while control is in the debugger (if it is reached), it is possible to continue program execution (thereby returning from the call to `cerror`) using the `continue` restart.

If *datum* is a condition, then that condition is used directly. In this case, the list of *arguments* need not be empty, but will be used only with the *continue-format-string* and will not be used to initialize *datum*.

If *datum* is a condition type (a class or class name), then the condition used is effectively the result of `(apply #'make-condition` *datum arguments*`)`.

If *datum* is a string, then the condition used is effectively the result of

```
(make-condition 'simple-error
                :format-string datum
                :format-arguments arguments)
```

The *continue-format-string* must be a string. Note that if *datum* is not a string, then the format arguments used by the *continue-format-string* will still be the list of *arguments* (which is in keyword format if *datum* is a condition type). In this case, some care may be necessary to set up the *continue-format-string* correctly. The `format` directive `~*`, which ignores and skips over `format` arguments, may be particularly useful in this situation.

The value returned by `cerror` is `nil`.

[Function] **signal** *datum* **&rest** *arguments*

Invokes the signal facility on a condition. If the condition is not handled, `signal` returns `nil`.

If *datum* is a condition, then that condition is used directly. In this case, it is an error for the list of *arguments* to be non-empty; that is, `error` must have been called with exactly one argument, the condition.

If *datum* is a condition type (a class or class name), then the condition used is effectively the result of `(apply #'make-condition` *datum arguments*`)`.

If *datum* is a string, then the condition used is effectively the result of

```
(make-condition 'simple-error
                :format-string datum
                :format-arguments arguments)
```

Note that if `(typep` *condition* `*break-on-signals*)` is true, then the debugger will be entered prior to beginning the process of signaling. The `continue` restart function may be used to continue with the signaling process; the restart is associated with the signaled condition as if by use of

`with-condition-restarts`. This is true also for all other functions and macros that signal conditions, such as `warn`, `error`, `cerror`, `assert`, and `check-type`.

During the dynamic extent of a call to `signal` with a particular condition, the effect of calling `signal` again on that condition object for a distinct abstract event is not defined. For example, although a handler *may* resignal a condition in order to allow outer handlers first shot at handling the condition, two distinct asynchronous keyboard events must not signal an the same (`eq`) condition object at the same time.

For further details about signaling and handling, see the discussion of condition handlers in section 28.3.17.

[Variable] ***break-on-signals***

This variable is intended primarily for use when the user is debugging programs that do signaling. The value of `*break-on-signals*` should be suitable as a second argument to `typep`, that is, a type or type specifier.

When `(typep` *`condition`* `*break-on-signals*)` is true, then calls to `signal` (and to other advertised functions such as `error` that implicitly call `signal`) will enter the debugger prior to signaling that *condition*. The `continue` restart may be used to continue with the normal signaling process; the restart is associated with the signaled condition as if by use of `with-condition-restarts`.

Note that `nil` is a valid type specifier. If the value of `*break-on-signals*` is `nil`, then `signal` will never enter the debugger in this implicit manner.

When setting this variable, the user is encouraged to choose the most restrictive specification that suffices. Setting this flag effectively violates the modular handling of condition signaling that this chapter seeks to establish. Its complete effect may be unpredictable in some cases, since the user may not be aware of the variety or number of calls to `signal` that are used in programs called only incidentally.

By default—and certainly in any "production" use—the value of this variable should be `nil`, both for reasons of performance and for reasons of modularity and abstraction.

X3J13 voted in March 1989 to remove `*break-on-warnings*` from the language; `*break-on-signals*` offers all the power of `*break-on-warnings*` and more.

28.4.2 Assertions

These facilities are designed to make it convenient for the user to insert error checks into code.

[Макрос] **check-type** place typespec [string]

[This supersedes the description of `check-type` given in section **??**.—GLS]

A `check-type` form signals an error of type `type-error` if the contents of *place* are not of the desired type.

If a condition is signaled, handlers of this condition can use the functions `type-error-datum` and `type-error-expected-type` to access the contents of *place* and the *typespec*, respectively.

This function can return only if the `store-value` restart is invoked, either explicitly from a handler or implicitly as one of the options offered by the debugger. The restart is associated with the signaled condition as if by use of `with-condition-restarts`.

If `store-value` is called, `check-type` will store the new value that is the argument to `store-value` (or that is prompted for interactively by the debugger) in *place* and start over, checking the type of the new value and signaling another error if it is still not the desired type. Subforms of *place* may be evaluated multiple times because of the implicit loop generated. `check-type` returns `nil`.

The *place* must be a generalized variable reference acceptable to `setf`. The *typespec* must be a type specifier; it is not evaluated. The `string` should be an English description of the type, starting with an indefinite article ("a" or "an"); it is evaluated. If the *string* is not supplied, it is computed automatically from the *typespec*. (The optional *string* argument is allowed because some applications of `check-type` may require a more specific description of what is wanted than can be generated automatically from the type specifier.)

The error message will mention the *place*, its contents, and the desired type.

Заметка для реализации: An implementation may choose to generate a somewhat differently worded error message if it recognizes that *place* is of a particular form, such as one of the arguments to the function that called `check-type`.

```
Lisp> (setq aardvarks '(sam harry fred))
 ⇒ (SAM HARRY FRED)
Lisp> (check-type aardvarks (array * (3)))
Error: The value of AARDVARKS, (SAM HARRY FRED),
       is not a 3-long array.
To continue, type :CONTINUE followed by an option number:
 1: Specify a value to use instead.
 2: Return to Lisp Toplevel.
Debug> :continue 1
Use Value: #(sam fred harry)
 ⇒ NIL
Lisp> aardvarks
 ⇒ #<ARRAY-3 13571>
Lisp> (map 'list #'identity aardvarks)
 ⇒ (SAM FRED HARRY)
Lisp> (setq aacount 'foo)
 ⇒ FOO
Lisp> (check-type aacount (integer 0 *) "a non-negative integer")
Error: The value of AACOUNT, FOO, is not a non-negative integer.
To continue, type :CONTINUE followed by an option number:
 1: Specify a value to use instead.
 2: Return to Lisp Toplevel.
Debug> :continue 2
Lisp>
```

[Макрос] **assert** test-form [({place}*) [datum {argument}*]]

[This supersedes the description of `assert` given in section **??**.—GLS]

An `assert` form signals an error if the value of the *test-form* is `nil`. Continuing from this error using the `continue` restart will allow the user to alter the values of some variables, and `assert` will then start over, evaluating the *test-form* again. (The restart is associated with the signaled condition as if by use of `with-condition-restarts`.) `assert` returns `nil`.

The *test-form* may be any form. Each *place* (there may be any number of them, or none) must be a generalized variable reference acceptable to `setf`. These should be variables on which *test-form* depends, whose values may sensibly be changed by the user in attempting to correct the error.

Subforms of each *place* are evaluated only if an error is signaled, and may be re-evaluated if the error is re-signaled (after continuing without actually fixing the problem).

The *datum* and *argument*s are evaluated only if an error is to be signaled, and re-evaluated if the error is to be signaled again.

If *datum* is a condition, then that condition is used directly. In this case, it is an error to specify any *argument*s.

If *datum* is a condition type (a class or class name), then the condition used is effectively the result of `(apply #'make-condition` *datum* `(list {`*argument*`}*))`.

If *datum* is a string, then the condition used is effectively the result of

```
(make-condition 'simple-error
                :format-string datum
                :format-arguments (list {argument}*))
```

If *datum* is omitted, then a condition of type `simple-error` is constructed using the *test-form* as data. For example, the following might be used:

```
(make-condition 'simple-error
  :format-string "The assertion ~S failed."
  :format-arguments '(test-form))
```

Note that the *test-form* itself, and not its value, is used as the format argument.

Заметка для реализации: The debugger need not include the *test-form* in the error message, and any *places* should not be included in the message, but they should be made available for the user's perusal. If the user gives the "continue" command, an opportunity should be presented to alter the values of any or all of the references. The details of this depend on the implementation's style of user interface, of course.

Here is an example of the use of `assert`:

```
(setq x (make-array '(3 5) :initial-element 3))
(setq y (make-array '(3 5) :initial-element 7))
```

```
(defun matrix-multiply (a b)
  (let ((*print-array* nil))
    (assert (and (= (array-rank a) (array-rank b) 2)
                 (= (array-dimension a 1)
                    (array-dimension b 0)))
            (a b)
            "Cannot multiply ~S by ~S." a b)
    (really-matrix-multiply a b)))

(matrix-multiply x y)
Error: Cannot multiply #<ARRAY-3-5 12345> by #<ARRAY-3-5 12364>.
To continue, type :CONTINUE followed by an option number:
 1: Specify new values.
 2: Return to Lisp Toplevel.
Debug> :continue 1
Value for A: x
Value for B: (make-array '(5 3) :initial-element 6)
 ⇒#2A((54 54 54 54 54)
          (54 54 54 54 54)
          (54 54 54 54 54)
          (54 54 54 54 54)
          (54 54 54 54 54))
```

28.4.3 Exhaustive Case Analysis

The syntax for `etypecase` and `ctypecase` is the same as for `typecase`, except that no `otherwise` clause is permitted. Similarly, the syntax for `ecase` and `ccase` is the same as for `case` except for the `otherwise` clause.

`etypecase` and `ecase` are similar to `typecase` and `case`, respectively, but signal a non-continuable error rather than returning `nil` if no clause is selected.

`ctypecase` and `ccase` are also similar to `typecase` and `case`, respectively, but signal a continuable error if no clause is selected.

[Макрос] **etypecase** keyform {(type {form}*)}*

[This supersedes the description of `etypecase` given in section **??**.—GLS]

This control construct is similar to `typecase`, but no explicit `otherwise` or `t` clause is permitted. If no clause is satisfied, `etypecase` signals an error (of type `type-error`) with a message constructed from the clauses. It is not permissible to continue from this error. To supply an error message, the user should use `typecase` with an `otherwise` clause containing a call to `error`. The name of this function stands for "exhaustive type case" or "error-checking type case."

Example:

```
Lisp> (setq x 1/3)
 ⇒ 1/3
Lisp> (etypecase x
        (integer (* x 4))
        (symbol (symbol-value x)))
Error: The value of X, 1/3, is neither an integer nor a symbol.
To continue, type :CONTINUE followed by an option number:
 1: Return to Lisp Toplevel.
Debug>
```

[Макрос] **ctypecase** keyplace {(type {form}*)}*

[This supersedes the description of `ctypecase` given in section **??**.—GLS]

This control construct is similar to `typecase`, but no explicit `otherwise` or `t` clause is permitted.

The *keyplace* must be a generalized variable reference acceptable to `setf`. If no clause is satisfied, `ctypecase` signals an error (of type `type-error`) with a message constructed from the clauses. This error may be continued using the `store-value` restart. The argument to `store-value` is stored in *keyplace* and then `ctypecase` starts over, making the type tests again. Subforms of *keyplace* may be evaluated multiple times. If the `store-value` restart is invoked interactively, the user will be prompted for the value to be used.

The name of this function is mnemonic for "continuable (exhaustive) type case."

Example:

```
Lisp> (setq x 1/3)
 ⇒ 1/3
Lisp> (ctypecase x
        (integer (* x 4))
        (symbol (symbol-value x)))
Error: The value of X, 1/3, is neither an integer nor a symbol.
To continue, type :CONTINUE followed by an option number:
 1: Specify a value to use instead.
 2: Return to Lisp Toplevel.
Debug> :continue 1
Use value: 3.7
Error: The value of X, 3.7, is neither an integer nor a symbol.
To continue, type :CONTINUE followed by an option number:
 1: Specify a value to use instead.
 2: Return to Lisp Toplevel.
Debug> :continue 1
Use value: 12
 ⇒ 48
```

[Макрос] **ecase** keyform {({({key}*) | key} {form}*)}*

[This supersedes the description of `ecase` given in section **??**.—GLS]

This control construct is similar to `case`, but no explicit `otherwise` or `t` clause is permitted. If no clause is satisfied, `ecase` signals an error (of type `type-error`) with a message constructed from the clauses. It is not permissible to continue from this error. To supply an error message, the user should use `case` with an `otherwise` clause containing a call to `error`. The name of this function stands for "exhaustive case" or "error-checking case."

Example:

```
Lisp> (setq x 1/3)
 ⇒ 1/3
Lisp> (ecase x
        (alpha (foo))
        (omega (bar))
        ((zeta phi) (baz)))
```

```
Error: The value of X, 1/3, is not ALPHA, OMEGA, ZETA, or PHI.
To continue, type :CONTINUE followed by an option number:
 1: Return to Lisp Toplevel.
Debug>
```

[Макрос] **ccase** keyplace {({({key}*) | key} {form}*)}*

[This supersedes the description of `ccase` given in section **??**.—GLS]

This control construct is similar to `case`, but no explicit `otherwise` or `t` clause is permitted.

The *keyplace* must be a generalized variable reference acceptable to `setf`. If no clause is satisfied, `ccase` signals an error (of type `type-error`) with a message constructed from the clauses. This error may be continued using the `store-value` restart. The argument to `store-value` is stored in *keyplace* and then `ccase` starts over, making the type tests again. Subforms of *keyplace* may be evaluated multiple times. If the `store-value` restart is invoked interactively, the user will be prompted for the value to be used.

The name of this function is mnemonic for "continuable (exhaustive) case."

Заметка для реализации: The `type-error` signaled by `ccase` and `ecase` is free to choose any representation of the acceptable argument type that it wishes for placement in the expected-type slot. It will always work to use type `(member . keys)`, but in some cases it may be more efficient, for example, to use a type that represents an integer subrange or a type composed using the `or` type specifier.

28.4.4 Handling Conditions

These macros allow a program to gain control when a condition is signaled.

[Макрос] **handler-case** expression {(typespec ([var]) {form}*)}*

Executes the given *expression* in a context where various specified handlers are active.

Each *typespec* may be any type specifier. If during the execution of the `expression` a condition is signaled for which there is an appropriate clause—that is, one for which `(typep` *`condition 'typespec`*`)` is true—and if there is no intervening handler for conditions of that type, then control is transferred

to the body of the relevant clause (unwinding the dynamic state appropriately in the process) and the given variable `var` is bound to the condition that was signaled. If no such condition is signaled and the computation runs to completion, then the values resulting from the `expression` are returned by the `handler-case` form.

If more than one case is provided, those cases are made accessible in parallel. That is, in

(handler-case *expression*
  (*type*$_1$ (*var*$_1$) *form*$_1$)
  (*type*$_2$ (*var*$_2$) *form*$_2$))

if the first clause (containing *form*$_1$) has been selected, the handler for the second is no longer visible (and vice versa).

The cases are searched sequentially from top to bottom. If a signaled condition matches more than one case (possible if there is type overlap) the earlier of the two cases will be selected.

If the variable *var* is not needed, it may be omitted. That is, a clause such as

(*type* (*var*) (declare (ignore *var*)) *form*)

may be written using the following shorthand notation:

(*type* () *form*)

If there are no forms in a selected case, the case returns `nil`. Note that

```
(handler-case expression
  (type1 (var1) . body1)
  (type2 (var2) . body2)
  ...)
```

is approximately equivalent to

```
(block #1=#:block-1
  (let (#2=#:var-2)
    (tagbody
      (handler-bind ((type1 #'(lambda (temp)
                                (setq #2# temp)
                                (go #3=#:tag-3)))
                     (type2 #'(lambda (temp)
                                (setq #2# temp)
                                (go #4=#:tag-4)))
                     ...)
        (return-from #1# expression))
      #3# (return-from #1# (let ((var1 #2#)) . body1))
      #4# (return-from #1# (let ((var2 #2#)) . body2))
      ...)))
```

[Note the use of "gensyms" such as `#:block-1` as block names, variables, and `tagbody` tags in this example, and the use of `#n=` and `#n#` read-macro syntax to indicate that the very same gensym appears in multiple places.—GLS]

As a special case, the *typespec* can also be the symbol `:no-error` in the last clause. If it is, it designates a clause that will take control if the *expression* returns normally. In that case, a completely general lambda-list may follow the symbol `:no-error`, and the arguments to which the lambda-list parameters are bound are like those for `multiple-value-call` on the return value of the *expression*. For example,

```
(handler-case expression
  (type1 (var1) . body1)
  (type2 (var2) . body2)
  ...
  (typen (varn) . bodyn)
  (:no-error (nvar1 nvar2 ... nvarm) . nbody))
```

is approximately equivalent to

```
(block #1=#:error-return
  (multiple-value-call #'(lambda (nvar1 nvar2 ... nvarm) . nbody)
    (block #2=#:normal-return
      (return-from #1#
        (handler-case (return-from #2# expression)
          (type1 (var1) . body1)
          (type2 (var2) . body2)
          ...
          (typen (varn) . bodyn))))))
```

Examples of the use of `handler-case`:

```
(handler-case (/ x y)
  (division-by-zero () nil))

(handler-case (open *the-file* :direction :input)
  (file-error (condition) (format t "~&Fooey: ~A~%" condition)))

(handler-case (some-user-function)
  (file-error (condition) condition)
  (division-by-zero () 0)
  ((or unbound-variable undefined-function) () 'unbound))

(handler-case (intern x y)
  (error (condition) condition)
  (:no-error (symbol status)
    (declare (ignore symbol))
    status))
```

[Макрос] **ignore-errors** {form}*

Executes its body in a context that handles conditions of type `error` by returning control to this form. If no such condition is signaled, any values returned by the last form are returned by `ignore-errors`. Otherwise, two values are returned: `nil` and the `error` condition that was signaled.

`ignore-errors` could be defined by

```
(defmacro ignore-errors (&body forms)
 `(handler-case (progn ,@forms)
    (error (c) (values nil c)))
```

[Макрос] **handler-bind** ({(typespec handler)}*) {form}*

Executes body in a dynamic context where the given handler bindings are in effect. Each *typespec* may be any type specifier. Each *handler* form should evaluate to a function to be used to handle conditions of the given type(s) during execution of the *form*s. This function should take a single argument, the condition being signaled.

If more than one binding is specified, the bindings are searched sequentially from top to bottom in search of a match (by visual analogy with `typecase`). If an appropriate *typespec* is found, the associated handler is run in a context where none of the handler bindings are visible (to avoid recursive errors). For example, in the case of

```
(handler-bind ((unbound-variable #'(lambda ...))
               (error #'(lambda ...)))
  ...)
```

if an unbound variable error is signaled in the body (and not handled by an intervening handler), the first function will be called. If any other kind of error is signaled, the second function will be called. In either case, neither handler will be active while executing the code in the associated function.

28.4.5 Defining Conditions

[The contents of this section are still a subject of some debate within X3J13. The reader may wish to take this section with a grain of salt, two aspirin tablets, and call a hacker in the morning.—GLS]

[Макрос] **define-condition** name ({parent-type}*)
[({slot-specifier}*) {option}*]

Defines a new condition type called *name*, which is a subtype of each given *parent-type*. Except as otherwise noted, the arguments are not evaluated.

Objects of this condition type will have all of the indicated *slot*s, plus any additional slots inherited from the parent types (its superclasses). If the *slot*s list is omitted, the empty list is assumed.

A *slot* must have the form

slot-specifier ::= *slot-name* | (*slot-name* [↓ *slot-option*])

For the syntax of a *slot-option*, see `defclass`. The slots of a condition object are normal CLOS slots. Note that `with-slots` may be used instead of accessor functions to access slots of a condition object.

`make-condition` will accept keywords (in the keyword package) with the print name of any of the designated slots, and will initialize the corresponding slots in conditions it creates.

Accessors are created according to the same rules as used by `defclass`.

The valid *options* are as follows:

`(:documentation` *`doc-string`*`)` The *doc-string* should be either `nil` or a string that describes the purpose of the condition type. If this option is omitted, `nil` is assumed. Calling `(documentation '`*`name`*` 'type)` will retrieve this information.

`(:report` *`exp`*`)` If *exp* is not a literal string, it must be a suitable argument to the `function` special operator. The expression `(function` *`exp`*`)` will be evaluated in the current lexical environment. It should produce a function of two arguments, a condition and a stream, that prints on the stream a description of the condition. This function is called whenever the condition is printed while `*print-escape*` is `nil`.

If *exp* is a literal string, it is shorthand for

```
(lambda (c s)
  (declare (ignore c))
  (write-string exp s))
```

[That is, a function is provided that will simply write the given string literally to the stream, regardless of the particular condition object supplied.—GLS]

The `:report` option is processed *after* the new condition type has been defined, so use of the slot accessors within the report function is permitted. If this option is not specified, information about how to report this type of condition will be inherited from the *parent-type*.

[X3J13 voted in March 1989 to integrate the Condition System and the Object System. In the original Condition System proposal, `define-condition` allowed only one *parent-type* (the inheritance structure was a simple hierarchy). Slot descriptions were much simpler, even simpler than those for `defstruct`:

slot ::= *slot-name* | (*slot-name*) | (*slot-name default-value*)

Similarly, `define-condition` allowed a `:conc-name` option similar to that of `defstruct`:

`(:conc-name` *`symbol-or-string`*`)` **Not now part of Common Lisp.** As with `defstruct`, this sets up automatic prefixing of the names of slot accessors. Also as in `defstruct`, the default behavior is to use the name of the new type, *name*, followed by a hyphen. (Generated names are interned in the package that is current at the time that the `define-condition` is processed).

One consequence of the vote was to make `define-condition` slot descriptions like those of `defclass`.—GLS]

Here are some examples of the use of `define-condition`.

The following form defines a condition of type `peg/hole-mismatch` that inherits from a condition type called `blocks-world-error`:

```
(define-condition peg/hole-mismatch (blocks-world-error)
                  (peg-shape hole-shape)
  (:report
```

```
(lambda (condition stream)
  (with-slots (peg-shape hole-shape) condition
    (format stream "A ~A peg cannot go in a ~A hole."
            peg-shape hole-shape))))
```

The new type has slots `peg-shape` and `hole-shape`, so `make-condition` will accept `:peg-shape` and `:hole-shape` keywords. The `with-slots` macro may be used to access the `peg-shape` and `hole-shape` slots, as illustrated in the `:report` information.

Here is another example. This defines a condition called `machine-error` that inherits from `error`:

```
(define-condition machine-error (error)
                  ((machine-name
                    :reader machine-error-machine-name))
  (:report (lambda (condition stream)
             (format stream "There is a problem with ~A."
                     (machine-error-machine-name condition)))))
```

Building on this definition, we can define a new error condition that is a subtype of `machine-error` for use when machines are not available:

```
(define-condition machine-not-available-error (machine-error) ()
  (:report (lambda (condition stream)
             (format stream "The machine ~A is not available."
                     (machine-error-machine-name condition)))))
```

We may now define a still more specific condition, built upon `machine-not-available-error`, that provides a default for `machine-name` but does not provide any new slots or report information. It just gives the `machine-name` slot a default initialization:

```
(define-condition my-favorite-machine-not-available-error
                  (machine-not-available-error)
                  ((machine-name :initform "MC.LCS.MIT.EDU")))
```

Note that since no `:report` clause was given, the information inherited from `machine-not-available-error` will be used to report this type of condition.

28.4.6 Creating Conditions

The function `make-condition` is the basic means for creating condition objects.

[Function] **make-condition** *type* **&rest** *slot-initializations*

Constructs a condition object of the given *type* using *slot-initializations* as a specification of the initial value of the slots. The newly created condition is returned.

The *slot-initializations* are alternating keyword/value pairs. For example:

```
(make-condition 'peg/hole-mismatch
                :peg-shape 'square :hole-shape 'round)
```

28.4.7 Establishing Restarts

The lowest-level form that creates restart points is called `restart-bind`. The `restart-case` macro is an abstraction that addresses many common needs for `restart-bind` while offering a more palatable syntax. See also

`with-simple-restart`. The function that transfers control to a restart point established by one of these macros is called `invoke-restart`.

All restarts have dynamic extent; a restart does not survive execution of the form that establishes it.

[Макрос] **with-simple-restart** (name format-string {format-argument}*) {form}*

This is shorthand for one of the most common uses of `restart-case`.

If the restart designated by *name* is not invoked while executing the *form*s, all values returned by the last *form* are returned. If that restart is invoked, control is transferred to the `with-simple-restart` form, which immediately returns the two values `nil` and `t`.

The *name* may be `nil`, in which case an anonymous restart is established.

`with-simple-restart` could be defined by

```
(defmacro with-simple-restart ((restart-name format-string
                                &rest format-arguments)
                               &body forms)
 `(restart-case (progn ,@forms)
    (,restart-name ()
      :report
        (lambda (stream)
          (format stream format-string ,@format-arguments))
      (values nil t))))
```

Here is an example of the use of `with-simple-restart`.

```
Lisp> (defun read-eval-print-loop (level)
        (with-simple-restart
            (abort "Exit command level ~D." level)
          (loop
            (with-simple-restart
                (abort "Return to command level ~D." level)
              (let ((form (prog2 (fresh-line)
                                 (read)
                                 (fresh-line))))
                (prin1 (eval form)))))))
⇒ READ-EVAL-PRINT-LOOP
```

```
Lisp> (read-eval-print-loop 1)
(+ 'a 3)
Error: The argument, A, to the function + was of the wrong type.
       The function expected a number.
To continue, type :CONTINUE followed by an option number:
 1: Specify a value to use this time.
 2: Return to command level 1.
 3: Exit command level 1.
 4: Return to Lisp Toplevel.
Debug>
```

Примечание: Some readers may wonder what ought to be done by the "abort" key (or whatever the implementation's interrupt key is—Control-C or Control-G, for example). Such interrupts, whether synchronous or asynchronous in nature, are beyond the scope of this chapter and indeed are not currently addressed by Common Lisp at all. This may be a topic worth standardizing under separate cover. Here is some speculation about some possible things that might happen.

An implementation might simply call `abort` or `break` directly without signaling any condition.

Another implementation might signal some condition related to the fact that a key had been pressed rather than to the action that should be taken. This is one way to allow user customization. Perhaps there would be an implementation-dependent `keyboard-interrupt` condition type with a slot containing the key that was pressed—or perhaps there would be such a condition type, but rather than its having slots, different subtypes of that type with names like `keyboard-abort`, `keyboard-break`, and so on might be signaled. That implementation would then document the action it would take if user programs failed to handle the condition, and perhaps ways for user programs to usefully dismiss the interrupt.

Заметка для реализации: Implementors are encouraged to make sure that there is always a restart named `abort` around any user code so that user code can call `abort` at any time and expect something reasonable to happen; exactly what the reasonable thing is may vary somewhat. Typically, in an interactive program, invoking `abort` should return the user to top level, though in some batch or multi-processing situations killing the running process might be more appropriate.

[Макрос] **restart-case** expression {(case-name arglist
{keyword value}*
{form}*)}*

The *expression* is evaluated in a dynamic context where the clauses have special meanings as points to which control may be transferred. If the *expression* finishes executing and returns any values, all such values are simply returned by the `restart-case` form. While the *expression* is running, any code may transfer control to one of the clauses (see `invoke-restart`). If a transfer occurs, the *form*s in the body of that clause will be evaluated and any values returned by the last such *form* will be returned by the `restart-case` form.

As a special case, if the *expression* is a list whose *car* is `signal`, `error`, `cerror`, or `warn`, then `with-condition-restarts` is implicitly used to associate the restarts with the condition to be signaled. For example,

```
(restart-case (signal weird-error)
  (become-confused ...)
  (rewind-line-printer ...)
  (halt-and-catch-fire ...))
```

is equivalent to

```
(restart-case (with-condition-restarts
                weird-error
                (list (find-restart 'become-confused)
                      (find-restart 'rewind-line-printer)
                      (find-restart 'halt-and-catch-fire))
                (signal weird-error))
  (become-confused ...)
  (rewind-line-printer ...)
  (halt-and-catch-fire ...))
```

If there are no *form*s in a selected clause, `restart-case` returns `nil`.

The *case-name* may be `nil` or a symbol naming this restart.

It is possible to have more than one clause use the same *case-name*. In this case, the first clause with that name will be found by `find-restart`.

The other clauses are accessible using `compute-restarts`. [In this respect, `restart-case` is rather different from `case`!—GLS]

Each *arglist* is a normal lambda-list containing parameters to be bound during the execution of its corresponding *form*s. These parameters are used to pass any necessary data from a call to `invoke-restart` to the `restart-case` clause.

By default, `invoke-restart-interactively` will pass no arguments and all parameters must be optional in order to accommodate interactive restarting. However, the parameters need not be optional if the `:interactive` keyword has been used to inform `invoke-restart-interactively` about how to compute a proper argument list.

The valid *keyword value* pairs are the following:

`:test` ***fn*** The *fn* must be a suitable argument for the `function` special operator. The expression `(function` ***fn*** `)` will be evaluated in the current lexical environment. It should produce a function of one argument, a condition. If this function returns `nil` when given some condition, functions such as `find-restart`, `compute-restart`, and `invoke-restart` will not consider this restart when searching for restarts associated with that condition. If this pair is not supplied, it is as if

```
(lambda (c) (declare (ignore c)) t)
```

were used for the *fn*.

`:interactive` ***fn*** The *fn* must be a suitable argument for the `function` special operator. The expression `(function` ***fn*** `)` will be evaluated in the current lexical environment. It should produce a function of no arguments that returns arguments to be used by `invoke-restart-interactively` when invoking this function. This function will be called in the dynamic environment available prior to any restart attempt. It may interact with the user on the stream in `*query-io*`.

If a restart is invoked interactively but no `:interactive` option was supplied, the argument list used in the invocation is the empty list.

`:report` ***exp*** If *exp* is not a literal string, it must be a suitable argument to the `function` special operator. The expression `(function` ***exp*** `)`

will be evaluated in the current lexical environment. It should produce a function of one argument, a stream, that prints on the stream a description of the restart. This function is called whenever the restart is printed while `*print-escape*` is `nil`.

If *exp* is a literal string, it is shorthand for

(lambda (s) (write-string *exp* s))

[That is, a function is provided that will simply write the given string literally to the stream.—GLS]

If a named restart is asked to report but no report information has been supplied, the name of the restart is used in generating default report text.

When `*print-escape*` is `nil`, the printer will use the report information for a restart. For example, a debugger might announce the action of typing "`:continue`" by executing the equivalent of

(format *debug-io* "~&~S – ~A~%" ':continue some-restart)

which might then display as something like

:CONTINUE – Return to command level.

It is an error if an unnamed restart is used and no report information is provided.

Обоснование: Unnamed restarts are required to have report information on the grounds that they are generally only useful interactively, and an interactive option that has no description is of little value.

Заметка для реализации: Implementations are encouraged to warn about this error at compilation time.

At run time, this error might be noticed when entering the debugger. Since signaling an error would probably cause recursive entry into the debugger (causing yet another recursive error, and so on), it is suggested that the debugger print some indication of such problems when they occur, but not actually signal errors.

Note that

(restart-case *expression*
  (*name*$_1$ *arglist*$_1$ *options*$_1$. *body*$_1$)
  (*name*$_2$ *arglist*$_2$ *options*$_2$. *body*$_2$)
  ...)

is essentially equivalent to

```
(block #1=#:block-1
  (let ((#2=#:var-2 nil))
    (tagbody
      (restart-bind ((name_1 #'(lambda (&rest temp)
                                 (setq #2# temp)
                                 (go #3=#:tag-3))
                               ⟨slightly transformed options_1⟩)
                     (name_2 #'(lambda (&rest temp)
                                 (setq #2# temp)
                                 (go #4=#:tag-4))
                               ⟨slightly transformed options_2⟩)
                     ...)
        (return-from #1# expression))
      #3# (return-from #1#
            (apply #'(lambda arglist_1 . body_1) #2#))
      #4# (return-from #1#
            (apply #'(lambda arglist_2 . body_2) #2#))
      ...)))
```

[Note the use of "gensyms" such as `#:block-1` as block names, variables, and `tagbody` tags in this example, and the use of `#n=` and `#n#` read-macro syntax to indicate that the very same gensym appears in multiple places.—GLS]

Here are some examples of the use of `restart-case`.

```
(loop
 (restart-case (return (apply function some-args))
  (new-function (new-function)
     :report "Use a different function."
     :interactive
       (lambda ()
         (list (prompt-for 'function "Function: ")))
    (setq function new-function))))

(loop
 (restart-case (return (apply function some-args))
  (nil (new-function)
     :report "Use a different function."
     :interactive
       (lambda ()
         (list (prompt-for 'function "Function: ")))
    (setq function new-function))))

(restart-case (a-command-loop)
 (return-from-command-level ()
    :report
      (lambda (s)     ;Argument s is a stream
        (format s "Return from command level ~D." level))
  nil))

(loop
 (restart-case (another-random-computation)
  (continue () nil)))
```

The first and second examples are equivalent from the point of view of someone using the interactive debugger, but they differ in one important aspect for non-interactive handling. If a handler "knows about" named restarts, as in, for example,

```
(when (find-restart 'new-function)
 (invoke-restart 'new-function the-replacement))
```

then only the first example, and not the second, will have control transferred to its correction clause, since only the first example uses a restart named `new-function`.

Here is a more complete example:

```
(let ((my-food 'milk)
      (my-color 'greenish-blue))
  (do ()
      ((not (bad-food-color-p my-food my-color)))
    (restart-case (error 'bad-food-color
                         :food my-food :color my-color)
      (use-food (new-food)
          :report "Use another food."
        (setq my-food new-food))
      (use-color (new-color)
          :report "Use another color."
        (setq my-color new-color))))
  ;; We won't get to here until MY-FOOD
  ;; and MY-COLOR are compatible.
  (list my-food my-color))
```

Assuming that `use-food` and `use-color` have been defined as

```
(defun use-food (new-food)
  (invoke-restart 'use-food new-food))

(defun use-color (new-color)
  (invoke-restart 'use-color new-color))
```

a handler can then restart from the error in either of two ways. It may correct the color or correct the food. For example:

```
#'(lambda (c) ... (use-color 'white) ...)   ;Corrects color

#'(lambda (c) ... (use-food 'cheese) ...)   ;Corrects food
```

Here is an example using `handler-bind` and `restart-case` that refers to a condition type `foo-error`, presumably defined elsewhere:

```
(handler-bind ((foo-error #'(lambda (ignore) (use-value 7))))
  (restart-case (error 'foo-error)
    (use-value (x) (* x x))))
⇒ 49
```

[Макрос] **restart-bind** ({(name function {keyword value}*)}*) {form}*

Executes a body of forms in a dynamic context where the given restart bindings are in effect.

Each *name* may be `nil` to indicate an anonymous restart, or some other symbol to indicate a named restart.

Each *function* is a form that should evaluate to a function to be used to perform the restart. If invoked, this function may either perform a non-local transfer of control or it may return normally. The function may take whatever arguments the programmer feels are appropriate; it will be invoked only if `invoke-restart` is used from a program, or if a user interactively asks the debugger to invoke it. In the case of interactive invocation, the `:interactive-function` option is used.

The valid *keyword value* pairs are as follows:

`:test-function` *`form`* The *form* will be evaluated in the current lexical environment and should return a function of one argument, a condition. If this function returns `nil` when given some condition, functions such as `find-restart`, `compute-restart`, and `invoke-restart` will not consider this restart when searching for restarts associated with that condition. If this pair is not supplied, it is as if

```
#'(lambda (c) (declare (ignore c)) t)
```

were used for the *form*.

`:interactive-function` *`form`* The *form* will be evaluated in the current lexical environment and should return a function of no arguments that constructs a list of arguments to be used by

`invoke-restart-interactively` when invoking this restart. The function may prompt interactively using `*query-io*` if necessary.

`:report-function` *`form`* The *form* will be evaluated in the current lexical environment and should return a function of one argument, a stream, that prints on the stream a summary of the action this restart will take. This function is called whenever the restart is printed while `*print-escape*` is `nil`.

[Макрос] **with-condition-restarts** condition-form restarts-form {declaration}* {form}*

The value of *condition-form* should be a condition C and the value of *restarts-form* should be a list of restarts (*`R1 R2 ...`*). The *form*s of the body are evaluated as an implicit `progn`. While in the dynamic context of the body, an attempt to find a restart associated with a particular condition C' will consider the restarts *R1*, *R2*, ... if C' is `eq` to C.

Usually this macro is not used explicitly in code, because `restart-case` handles most of the common uses in a way that is syntactically more concise.

[The X3J13 vote left it unclear whether `with-condition-restarts` permits declarations to appear at the heads of its body. I believe that was the intent, but this is only my interpretation.—GLS]

28.4.8 Finding and Manipulating Restarts

The following functions determine what restarts are active and invoke restarts.

[Function] **compute-restarts** **&optional** *condition*

Uses the dynamic state of the program to compute a list of the restarts that are currently active. See `restart-bind`.

If *condition* is `nil` or not supplied, all outstanding restarts are returned. If *condition* is not `nil`, only restarts associated with that condition are returned.

Each restart represents a function that can be called to perform some form of recovery action, usually a transfer of control to an outer point in the running program. Implementations are free to implement these objects

in whatever manner is most convenient; the objects need have only dynamic extent (relative to the scope of the binding form that instantiates them).

The list that results from a call to `compute-restarts` is ordered so that the inner (that is, more recently established) restarts are nearer the head of the list.

Note, too, that `compute-restarts` returns all valid restarts, including anonymous ones, even if some of them have the same name as others and would therefore not be found by `find-restart` when given a symbol argument.

Implementations are permitted, but not required, to return different (that is, non-`eq`) lists from repeated calls to `compute-restarts` while in the same dynamic environment. It is an error to modify the list that is returned by `compute-restarts`.

[Function] **restart-name** *restart*

Returns the name of the given *restart*, or `nil` if it is not named.

[Function] **find-restart** *restart-identifier* **&optional** *condition*

Searches for a particular restart in the current dynamic environment.

If *condition* is `nil` or not supplied, all outstanding restarts are considered. If *condition* is not `nil`, only restarts associated with that condition are considered.

If the *restart-identifier* is a non-`nil` symbol, then the innermost (that is, most recently established) restart with that name is returned; `nil` is returned if no such restart is found.

If *restart-identifier* is a restart object, then it is simply returned, unless it is not currently active, in which case `nil` is returned.

Although anonymous restarts have a name of `nil`, it is an error for the symbol `nil` to be given as the *restart-identifier*. Applications that would seem to require this should be rewritten to make appropriate use of `compute-restarts` instead.

[Function] **invoke-restart** *restart-identifier* **&rest** *arguments*

Calls the function associated with the given *restart-identifier*, passing any given *arguments*. The *restart-identifier* must be a restart or the non-

null name of a restart that is valid in the current dynamic context. If the argument is not valid, an error of type `control-error` will be signaled.

Заметка для реализации: Restart functions call this function, not vice versa.

[Function] **invoke-restart-interactively** *restart-identifier*

Calls the function associated with the given *restart-identifier*, prompting for any necessary arguments. The *restart-identifier* must be a restart or the non-null name of a restart that is valid in the current dynamic context. If the argument is not valid, an error of type `control-error` will be signaled.

The function `invoke-restart-interactively` will prompt for arguments by executing the code provided in the `:interactive` keyword to `restart-case` or `:interactive-function` keyword to `restart-bind`.

If no `:interactive` or `:interactive-function` option has been supplied in the corresponding `restart-case` or `restart-bind`, then it is an error if the restart takes required arguments. If the arguments are optional, an empty argument list will be used in this case.

Once `invoke-restart-interactively` has calculated the arguments, it simply performs `(apply #'invoke-restart` *`restart-identifier arguments`*`)`.

`invoke-restart-interactively` is used internally by the debugger and may also be useful in implementing other portable, interactive debugging tools.

28.4.9 Warnings

Warnings are a subclass of errors that are conventionally regarded as "mild."

[Function] **warn** *datum* **&rest** *arguments*

[This supersedes the description of `warn` given in section **??**.—GLS]

Warns about a situation, by signaling a condition of type `warning`.

If *datum* is a condition, then that condition is used directly. In this case, if the condition is not of type `warning` or arguments is non-`nil`, an error of type `type-error` is signaled.

If *datum* is a condition type (a class or class name), then the condition used is effectively the result of `(apply #'make-condition` *`datum`*

arguments). This result must be of type `warning` or an error of type `type-error` is signaled.

If *datum* is a string, then the condition used is effectively the result of

```
(make-condition 'simple-error
                :format-string datum
                :format-arguments arguments)
```

The precise mechanism for warning is as follows.

1. The warning condition is signaled.

   While the `warning` condition is being signaled, the `muffle-warning` restart is established for use by a handler to bypass further action by `warn` (that is, to cause `warn` to immediately return `nil`).

   As part of the signaling process, if `(typep` *condition* `*break-on-signals*)` is true, then a `break` will occur prior to beginning the signaling process.

2. If no handlers for the warning condition are found, or if all such handlers decline, then the condition will be reported to `*error-output*` by the `warn` function (with possible implementation-specific extra output such as motion to a fresh line before or after the display of the warning, or supplying some introductory text mentioning the name of the function that called `warn` or the fact that this is a warning).

3. The value returned by `warn` (if it returns) is `nil`.

28.4.10 Restart Functions

Common Lisp has the following restart functions built in.

[Function] **abort** **&optional** *condition*

This function transfers control to the restart named `abort`. If no such restart exists, `abort` signals an error of type `control-error`.

If *condition* is `nil` or not supplied, all outstanding restarts are considered. If *condition* is not `nil`, only restarts associated with that condition are considered.

The purpose of the `abort` restart is generally to allow control to return to the innermost "command level."

[Function] **continue** **&optional** *condition*

This function transfers control to the restart named `continue`. If no such restart exists, `continue` returns `nil`.

If *condition* is `nil` or not supplied, all outstanding restarts are considered. If *condition* is not `nil`, only restarts associated with that condition are considered.

The `continue` restart is generally part of simple protocols where there is a single "obvious" way to continue, as with `break` and `cerror`. Some user-defined protocols may also wish to incorporate it for similar reasons. In general, however, it is more reliable to design a special-purpose restart with a name that better suits the particular application.

[Function] **muffle-warning** **&optional** *condition*

This function transfers control to the restart named `muffle-warning`. If no such restart exists, `muffle-warning` signals an error of type `control-error`.

If *condition* is `nil` or not supplied, all outstanding restarts are considered. If *condition* is not `nil`, only restarts associated with that condition are considered.

`warn` sets up this restart so that handlers of `warning` conditions have a way to tell `warn` that a `warning` has already been dealt with and that no further action is warranted.

[Function] **store-value** *value* **&optional** *condition*

This function transfers control (and one value) to the restart named `store-value`. If no such restart exists, `store-value` returns `nil`.

If *condition* is `nil` or not supplied, all outstanding restarts are considered. If *condition* is not `nil`, only restarts associated with that condition are considered.

The `store-value` restart is generally used by handlers trying to recover from errors of types such as `cell-error` or `type-error`, where the handler may wish to supply a replacement datum to be stored permanently.

[Function] **use-value** *value* **&optional** *condition*

This function transfers control (and one value) to the restart named `use-value`. If no such restart exists, `use-value` returns `nil`.

If *condition* is `nil` or not supplied, all outstanding restarts are considered. If *condition* is not `nil`, only restarts associated with that condition are considered.

The `use-value` restart is generally used by handlers trying to recover from errors of types such as `cell-error`, where the handler may wish to supply a replacement datum for one-time use.

28.4.11 Debugging Utilities

Common Lisp does not specify exactly what a debugger is or does, but it does provide certain means for indicating intent to transfer control to a supervisory or debugging facility.

[Function] **break** **&optional** *format-string* **&rest** *format-arguments*

[This supersedes the description of `break` given in section **??**.—GLS]

The function `break` prints the message described by the *format-string* and *format-arguments* and then goes directly into the debugger without allowing any possibility of interception by programmed error-handling facilities.

If no *format-string* is supplied, a suitable default will be generated.

If continued, `break` returns `nil`.

Note that `break` is presumed to be used as a way of inserting temporary debugging "breakpoints" in a program, not as a way of signaling errors; it is expected that continuing from a `break` will not trigger any unusual recovery action. For this reason, `break` does not take the additional format control string that `cerror` takes as its first argument. This and the lack of any possibility of interception by programmed error handling are the only program-visible differences between `break` and `cerror`. The user interface aspects of these functions are permitted to vary more widely; for example, it

is permissible for a read-eval-print loop to be entered by `break` rather than by the conventional debugger.

`break` could be defined by

```
(defun break (&optional (format-string "Break")
              &rest format-arguments)
  (with-simple-restart (continue "Return from BREAK.")
    (invoke-debugger
      (make-condition 'simple-condition
                      :format-string format-string
                      :format-arguments format-arguments)))
  nil)
```

[Function] **invoke-debugger** *condition*

Attempts interactive handling of its argument, which must be a condition.

If the variable `*debugger-hook*` is not `nil`, it will be called as a function on two arguments: the *condition* being handled and the value of `*debugger-hook*`. If a hook function returns normally, the standard debugger will be tried.

The standard debugger will never directly return. Return can occur only by a special transfer of control, such as the use of a restart.

Примечание: The exact way in which the debugger interacts with users is expected to vary considerably from system to system. For example, some systems may use a keyboard interface, while others may use a mouse interface. Of those systems using keyboard commands, some may use single-character commands and others may use parsed line-at-a-time commands. The exact set of commands will vary as well. The important properties of a debugger are that it makes information about the error accessible and that it makes the set of apparent restarts easily accessible.

It is desirable to have a mode where the debugger allows other features, such as the ability to inspect data, stacks, etc. However, it may sometimes be appropriate to have this kind of information hidden from users. Experience on the Lisp Machines has shown that some users who are not programmers develop a terrible phobia of debuggers. The reason for this usually may be traced to the fact that the debugger is very foreign to them and provides an overwhelming amount of information of interest only to programmers. With the advent of restarts, there is a clear mechanism for the construction of “friendly” debuggers. Programmers can be taught how to get to the information they need for debugging, but it should be

possible to construct user interfaces to the debugger that are natural, convenient, intelligible, and friendly even to non-programmers.

[Variable] ***debugger-hook***

This variable should hold either `nil` or a function of two arguments, a condition and the value of `*debugger-hook*`. This function may either handle the condition (transfer control) or return normally (allowing the standard debugger to run).

Note that, to minimize recursive errors while debugging, `*debugger-hook*` is bound to `nil` when calling this function. When evaluating code typed in by the user interactively, the hook function may want to bind `*debugger-hook*` to the function that was its second argument so that recursive errors can be handled using the same interactive facility.

28.5 Predefined Condition Types

[The proposal for the Common Lisp Condition System introduced a new notation for documenting types, treating them in the same syntactic manner as functions and variables. This notation is used in this section but is not reflected throughout the entire book.—GLS]

X3J13 voted in March 1989 to integrate the Condition System and the Object System. All condition types are CLOS classes and all condition objects are ordinary CLOS objects.

[Type] **restart**

This is the data type used to represent a restart.

The Common Lisp condition type hierarchy is illustrated in table 28.1.

The types that are not leaves in the hierarchy (that is, `condition`, `warning`, `storage-condition`, `error`, `arithmetic-error`, `control-error`, and so on) are provided primarily for type inclusion purposes. Normally they would not be directly instantiated.

Implementations are permitted to support non-portable synonyms for these types, as well as to introduce other types that are above, below, or between the types shown in this tree as long as the indicated subtype relationships are not violated.

The types `simple-condition`, `serious-condition`, and `warning` are pairwise disjoint. The type `error` is also disjoint from types `simple-condition` and `warning`.

[Type] **condition**

All types of conditions, whether error or non-error, must inherit from this type.

[Type] **warning**

All types of warnings should inherit from this type. This is a subtype of `condition`.

[Type] **serious-condition**

All serious conditions (conditions serious enough to require interactive intervention if not handled) should inherit from this type. This is a subtype of `condition`.

This condition type is provided primarily for terminological convenience. In fact, signaling a condition that inherits from `serious-condition` does not force entry into the debugger. Rather, it is conventional to use `error` (or something built on `error`) to signal conditions that are of this type, and to use `signal` to signal conditions that are not of this type.

[Type] **error**

All types of error conditions inherit from this condition. This is a subtype of `serious-condition`.

The default condition type for `signal` and `warn` is `simple-condition`. The default condition type for `error` and `cerror` is `simple-error`.

[Type] **simple-condition**

Conditions signaled by `signal` when given a format string as a first argument are of this type. This is a subtype of `condition`. The initialization keywords `:format-string` and `:format-arguments` are supported to initialize the slots, which can be accessed using `simple-condition-format-control` and `simple-condition-format-arguments`. If `:format-arguments` is not supplied to `make-condition`, the format-arguments slot defaults to `nil`.

[Type] **simple-warning**

Conditions signaled by `warn` when given a format string as a first argument are of this type. This is a subtype of `warning`. The initialization keywords `:format-string` and `:format-arguments` are supported to initialize the slots, which can be accessed using `simple-condition-format-control` and `simple-condition-format-arguments`. If `:format-arguments` is not supplied to `make-condition`, the format-arguments slot defaults to `nil`.

In implementations supporting multiple inheritance, this type will also be a subtype of `simple-condition`.

[Type] **simple-error**

Conditions signaled by `error` and `cerror` when given a format string as a first argument are of this type. This is a subtype of `error`. The initialization keywords `:format-string` and `:format-arguments` are supported to initialize the slots, which can be accessed using `simple-condition-format-control` and `simple-condition-format-arguments`. If `:format-arguments` is not supplied to `make-condition`, the format-arguments slot defaults to `nil`.

In implementations supporting multiple inheritance, this type will also be a subtype of `simple-condition`.

[Function] **simple-condition-format-control** *condition*

Accesses the format-string slot of a given *condition*, which must be of type `simple-condition`, `simple-warning`, `simple-error`, or `simple-type-error`.

[Function] **simple-condition-format-arguments** *condition*

Accesses the format-arguments slot of a given *condition*, which must be of type `simple-condition`, `simple-warning`, `simple-error`, or `simple-type-error`.

[Type] **storage-condition**

Conditions that relate to storage overflow should inherit from this type. This is a subtype of `serious-condition`.

[Type] **type-error**

Errors in the transfer of data in a program should inherit from this type. This is a subtype of `error`. For example, conditions to be signaled by `check-type` should inherit from this type. The initialization keywords `:datum` and `:expected-type` are supported to initialize the slots, which can be accessed using `type-error-datum` and `type-error-expected-type`.

[Function] **type-error-datum** *condition*

Accesses the datum slot of a given *condition*, which must be of type `type-error`.

[Function] **type-error-expected-type** *condition*

Accesses the expected-type slot of a given *condition*, which must be of type `type-error`. Users of `type-error` conditions are expected to fill this slot with an object that is a valid Common Lisp type specifier.

[Type] **simple-type-error**

Conditions signaled by facilities similar to `check-type` may want to use this type. The initialization keywords `:format-string` and `:format-arguments` are supported to initialize the slots, which can be accessed using `simple-condition-format-control` and `simple-condition-format-arguments`. If `:format-arguments` is not supplied to `make-condition`, the format-arguments slot defaults to `nil`.

In implementations supporting multiple inheritance, this type will also be a subtype of `simple-condition`.

[Type] **program-error**

Errors relating to incorrect program syntax that are statically detectable should inherit from this type (regardless of whether they are in fact statically detected). This is a subtype of `error`. This is *not* a subtype of `control-error`.

[Type] **control-error**

Errors in the dynamic transfer of control in a program should inherit from this type. This is a subtype of `error`. This is *not* a subtype of `program-error`.

The errors that result from giving `throw` a tag that is not active or from giving `go` or `return-from` a tag that is no longer dynamically available are control errors.

On the other hand, the errors that result from naming a `go` tag or `return-from` tag that is not lexically apparent are not control errors. They are program errors. See `program-error`.

[Type] **package-error**

Errors that occur during operations on packages should inherit from this type. This is a subtype of `error`. The initialization keyword `:package` is supported to initialize the slot, which can be accessed using `package-error-package`.

[Function] **package-error-package** *condition*

Accesses the package (or package name) that was being modified or manipulated in a *condition* of type `package-error`.

[Type] **stream-error**

Errors that occur during input from, output to, or closing a stream should inherit from this type. This is a subtype of `error`. The initialization keyword `:stream` is supported to initialize the slot, which can be accessed using `stream-error-stream`.

[Function] **stream-error-stream** *condition*

Accesses the offending stream of a *condition* of type `stream-error`.

[Type] **end-of-file**

The error that results when a read operation is done on a stream that has no more tokens or characters should inherit from this type. This is a subtype of `stream-error`.

[Type] **file-error**

Errors that occur during an attempt to open a file, or during some low-level transaction with a file system, should inherit from this type. This is

a subtype of `error`. The initialization keyword `:pathname` is supported to initialize the slot, which can be accessed using `file-error-pathname`.

[Function] **file-error-pathname** *condition*

Accesses the offending pathname of a *condition* of type `file-error`.

[Type] **cell-error**

Errors that occur while accessing a location should inherit from this type. This is a subtype of `error`. The initialization keyword `:name` is supported to initialize the slot, which can be accessed using `cell-error-name`.

[Function] **cell-error-name** *condition*

Accesses the offending cell name of a *condition* of type `cell-error`.

[Type] **unbound-variable**

The error that results from trying to access the value of an unbound variable should inherit from this type. This is a subtype of `cell-error`.

[Type] **undefined-function**

The error that results from trying to access the value of an undefined function should inherit from this type. This is a subtype of `cell-error`.

Примечание: [Note: This remark was written well before the vote by X3J13 in June 1988 to add the Common Lisp Object System to the forthcoming draft standard (see chapter 27) and the vote to integrate the Condition System and the Object System. I have retained the remark here for reasons of historical interest.—GLS]

Some readers may wonder why `undefined-function` is not defined to inherit from some condition such as `control-error`. The answer is that any such arrangement would require the presence of multiple inheritance—a luxury we do not currently have (without resorting to `deftype`, which we are currently avoiding). When the Common Lisp Object System comes into being, we might want to consider issues like this. Multiple inheritance makes a lot of things in a condition system much more flexible to deal with.

[Type] **arithmetic-error**

Errors that occur while doing arithmetic type operations should inherit from this type. This is a subtype of `error`. The initialization keywords `:operation` and `:operands` are supported to initialize the slots, which can be accessed using `arithmetic-error-operation` and `arithmetic-error-operands`.

[Function] **arithmetic-error-operation** *condition*

Accesses the offending operation of a condition of type `arithmetic-error`.

[Function] **arithmetic-error-operands** *condition*

Accesses a list of the offending operands in a condition of type `arithmetic-error`.

[Type] **division-by-zero**

Errors that occur because of division by zero should inherit from this type. This is a subtype of `arithmetic-error`.

[Type] **floating-point-overflow**

Errors that occur because of floating-point overflow should inherit from this type. This is a subtype of `arithmetic-error`.

[Type] **floating-point-underflow**

Errors that occur because of floating-point underflow should inherit from this type. This is a subtype of `arithmetic-error`.

Таблица 28.1: Condition Type Hierarchy

```
condition
    simple-condition
    serious-condition
        error
            simple-error
            arithmetic-error
                division-by-zero
                floating-point-overflow
                floating-point-underflow
                ...
            cell-error
                unbound-variable
                undefined-function
                ...
            control-error
            file-error
            package-error
            program-error
            stream-error
                end-of-file
                ...
            type-error
                simple-type-error
                ...
            ...
        storage-condition
        ...
    warning
        simple-warning
        ...
    ...
```

Библиография

[1] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison-Wesley (Reading, Massachusetts, 1985).

[2] Alberga, Cyril N., Bosman-Clark, Chris, Mikelsons, Martin, Van Deusen, Mary S., and Padget, Julian. Experience with an uncommon Lisp. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*. ACM SIGPLAN/SIGACT/SIGART (Cambridge, Massachusetts, August 1986), 39–53.

[3] *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978 edition. American National Standards Institute, Inc. (New York, 1978).

[4] Bates, Raymond L., Dyer, David, and Feber, Mark. Recent developments in ISI-Interlisp. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*. ACM SIGPLAN/SIGACT/SIGART (Austin, Texas, August 1984), 129–139.

[5] Bobrow, Daniel G., DiMichiel, Linda G., Gabriel, Richard P., Keene, Sonya E., Kiczales, Gregor, and Moon, David A. Common Lisp Object System Specification: X3J13 Document 88-002R. *SIGPLAN Notices 23* (September 1988).

[6] Bobrow, Daniel G., DiMichiel, Linda G., Gabriel, Richard P., Keene, Sonya E., Kiczales, Gregor, and Moon, David A. Common Lisp Object System specification: 1. Programmer interface concepts. *Lisp and Symbolic Computation 1*, 3/4 (January 1989), 245–298.

[7] Bobrow, Daniel G., DiMichiel, Linda G., Gabriel, Richard P., Keene, Sonya E., Kiczales, Gregor, and Moon, David A. Common Lisp Object

System specification: 2. Functions in the programmer interface. *Lisp and Symbolic Computation 1*, 3/4 (January 1989), 299–394.

[8] Bobrow, Daniel G., and Kiczales, Gregor. The Common Lisp Object System metaobject kernel: A status report. In *Proc. 1988 ACM Conference on Lisp and Functional Programming*. ACM SIGPLAN/SIGACT/SIGART (Snowbird, Utah, July 1988), 309–315.

[9] Brooks, Rodney A., and Gabriel, Richard P. A critique of Common Lisp. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*. ACM SIGPLAN/SIGACT/SIGART (Austin, Texas, August 1984), 1–8.

[10] Brooks, Rodney A., Gabriel, Richard P., and Steele, Guy L., Jr. S-1 Common Lisp implementation. In *Proc. 1982 ACM Symposium on Lisp and Functional Programming*. ACM SIGPLAN/SIGACT/SIGART (Pittsburgh, Pennsylvania, August 1982), 108–113.

[11] Brooks, Rodney A., Gabriel, Richard P., and Steele, Guy L., Jr. An optimizing compiler for lexically scoped lisp. In *Proc. 1982 Symposium on Compiler Construction*. ACM SIGPLAN (Boston, June 1982), 261–275. Proceedings published as *ACM SIGPLAN Notices 17*, 6 (June 1982).

[12] Clinger, William (ed.) *The Revised Revised Report on Scheme; or, An Uncommon Lisp*. AI Memo 848. MIT Artificial Intelligence Laboratory (Cambridge, Massachusetts, August 1985).

[13] Clinger, William (ed.) *The Revised Revised Report on Scheme; or, An Uncommon Lisp*. Computer Science Department Technical Report 174. Indiana University (Bloomington, Indiana, June 1985).

[14] Cody, William J., Jr., and Waite, William. *Software Manual for the Elementary Functions*. Prentice-Hall (Englewood Cliffs, New Jersey, 1980).

[15] Committee, ANSI X3J3. Draft proposed American National Standard Fortran. *ACM SIGPLAN Notices 11*, 3 (March 1976).

[16] Coonen, Jerome T. Errata for "An implementation guide to a proposed standard for floating-point arithmetic." *Computer 14*, 3 (March 1981), 62. These are errata for [17].

[17] Coonen, Jerome T. An implementation guide to a proposed standard for floating-point arithmetic. *Computer 13*, 1 (January 1980), 68–79. Errata for this paper appeared as [16].

[18] DiMichiel, Linda G. Overview: The Common Lisp Object System. *Lisp and Symbolic Computation 1*, 3/4 (January 1989), 227–244.

[19] Fateman, Richard J. Reply to an editorial. *ACM SIGSAM Bulletin 25* (March 1973), 9–11.

[20] Goldberg, Adele, and Robson, David. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley (Reading, Massachusetts, 1983).

[21] Griss, Martin L., Benson, Eric, and Hearn, Anthony C. Current status of a portable LISP compiler. In *Proc. 1982 Symposium on Compiler Construction*. ACM SIGPLAN (Boston, June 1982), 276–283. Proceedings published as *ACM SIGPLAN Notices 17*, 6 (June 1982).

[22] Harrenstien, Kenneth L. *Time Server*. Request for Comments (RFC) 738 (NIC 42218). ARPANET Network Working Group (October 1977). Available from the ARPANET Network Information Center.

[23] IEEE Computer Society Standard Committee, Floating-Point Working Group, Microprocessor Standards Subcommittee. A proposed standard for binary floating-point arithmetic. *Computer 14*, 3 (March 1981), 51–62.

[24] ISO. *Information Processing—Coded Character Sets for Text Communication, Part 2: Latin Alphabetic and Non-alphabetic Graphic Characters*. ISO (1983).

[25] Kahan, W. Branch cuts for complex elementary functions; or, Much ado about nothing's sign bit. In Iserles, A., and Powell, M. (eds.), *The State of the Art in Numerical Analysis*. Clarendon Press (1987), 165–211.

[26] Keene, Sonya E. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley (Reading, Massachusetts, 1989).

[27] Knuth, Donald E. *Seminumerical Algorithms*. Volume 2 of *The Art of Computer Programming*. Addison-Wesley (Reading, Massachusetts, 1969).

[28] Knuth, Donald E. *The TEXbook*. Volume A of *Computers and Typesetting*. Addison-Wesley (Reading, Massachusetts, 1986).

[29] Knuth, Donald E. *TEX: The Program*. Volume B of *Computers and Typesetting*. Addison-Wesley (Reading, Massachusetts, 1986).

[30] Lamport, Leslie. *LATEX: A Document Preparation System*. Addison-Wesley (Reading, Massachusetts, 1986).

[31] Marti, J., Hearn, A. C., Griss, M. L., and Griss, C. Standard Lisp report. *ACM SIGPLAN Notices 14*, 10 (October 1979), 48–68.

[32] McDonnell, E. E. The story of ∘. *APL Quote Quad 8*, 2 (December 1977), 48–54.

[33] Moon, David. *MacLISP Reference Manual, Revision 0*. MIT Project MAC (Cambridge, Massachusetts, April 1974).

[34] Moon, David; Stallman, Richard; and Weinreb, Daniel. *LISP Machine Manual, Fifth Edition*. MIT Artificial Intelligence Laboratory (Cambridge, Massachusetts, January 1983).

[35] Padget, Julian, et al. Desiderata for the standardisation of Lisp. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*. ACM SIGPLAN/SIGACT/SIGART (Cambridge, Massachusetts, August 1986), 54–66.

[36] Penfield, Paul, Jr. Principal values and branch cuts in complex APL. In *APL 81 Conference Proceedings*. ACM SIGAPL (San Francisco, September 1981), 248–256. Proceedings published as *APL Quote Quad 12*, 1 (September 1981).

[37] Pitman, Kent M. *The Revised MacLISP Manual*. MIT/LCS/TR 295. MIT Laboratory for Computer Science (Cambridge, Massachusetts, May 1983).

[38] Pitman, Kent M. *Exceptional Situations in Lisp*. Working paper 268. MIT Artificial Intelligence Laboratory (Cambridge, Massachusetts).

[39] Queinnec, Christian, and Cointe, Pierre. An open-ended data representation model for EU_LISP. In *Proc. 1988 ACM Conference on Lisp and*

Functional Programming. ACM SIGPLAN/SIGACT/SIGART (Snowbird, Utah, July 1988), 298–308.

[40] Rees, Jonathan, Clinger, William, et al. Revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices 21*, 12 (December 1986), 37–79.

[41] Reiser, John F. *Analysis of Additive Random Number Generators*. Technical Report STAN-CS-77-601. Stanford University Computer Science Department (Palo Alto, California, March 1977).

[42] Roylance, Gerald. Expressing mathematical subroutines constructively. In *Proc. 1988 ACM Conference on Lisp and Functional Programming*. ACM SIGPLAN/SIGACT/SIGART (Snowbird, Utah, July 1988), 8–13.

[43] Steele, Guy L., Jr. An overview of Common Lisp. In *Proc. 1982 ACM Symposium on Lisp and Functional Programming*. ACM SIGPLAN/SIGACT/SIGART (Pittsburgh, Pennsylvania, August 1982), 98–107.

[44] Steele, Guy L., Jr., and Hillis, W. Daniel. Connection Machine Lisp: Fine-grained parallel symbolic processing. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*. ACM SIGPLAN/SIGACT/SIGART (Cambridge, Massachusetts, August 1986), 279–297.

[45] Steele, Guy Lewis, Jr. *RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization)*. Technical Report 474. MIT Artificial Intelligence Laboratory (Cambridge, Massachusetts, May 1978).

[46] Steele, Guy Lewis, Jr., and Sussman, Gerald Jay. *The Revised Report on SCHEME: A Dialect of LISP*. AI Memo 452. MIT Artificial Intelligence Laboratory (Cambridge, Massachusetts, January 1978).

[47] Suzuki, Norihisa. Analysis of pointer "rotation". *Communications of the ACM 25*, 5 (May 1982), 330–335.

[48] Swanson, Mark, Kessler, Robert, and Lindstrom, Gary. An implementation of Portable Standard Lisp on the BBN Butterfly. In *Proc. 1988 ACM Conference on Lisp and Functional Programming*. ACM SIGPLAN/SIGACT/SIGART (Snowbird, Utah, July 1988), 132–142.

[49] Symbolics, Inc. *Signalling and Handling Conditions.* (Cambridge, Massachusetts, 1983).

[50] Teitelman, Warren, et al. *InterLISP Reference Manual.* Xerox Palo Alto Research Center (Palo Alto, California, 1978). Third revision.

[51] The Utah Symbolic Computation Group. *The Portable Standard LISP Users Manual.* Technical Report TR-10. Department of Computer Science, University of Utah (Salt Lake City, Utah, January 1982).

[52] Waters, Richard C. *Optimization of Series Expressions, Part I: User's Manual for the Series Macro Package.* AI Memo 1082. MIT Artificial Intelligence Laboratory (Cambridge, Massachusetts, January 1989).

[53] Waters, Richard C. *Optimization of Series Expressions, Part II: Overview of the Theory and Implementation.* AI Memo 1083. MIT Artificial Intelligence Laboratory (Cambridge, Massachusetts, January 1989).

[54] Waters, Richard C. *XP: A Common Lisp Pretty Printing System.* AI Memo 1102. MIT Artificial Intelligence Laboratory (Cambridge, Massachusetts, March 1989).

[55] Weinreb, Daniel, and Moon, David. *LISP Machine Manual, Fourth Edition.* MIT Artificial Intelligence Laboratory (Cambridge, Massachusetts, July 1981).

[56] Wholey, Skef, and Fahlman, Scott E. The design of an instruction set for Common Lisp. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming.* ACM SIGPLAN/SIGACT/SIGART (Austin, Texas, August 1984), 150–158.

[57] Wholey, Skef, and Steele, Guy L., Jr. Connection Machine Lisp: A dialect of Common Lisp for data parallel programming. In Kartashev, Lana P., and Kartashev, Steven I. (eds.), *Proc. Second International Conference on Supercomputing.* Volume III. International Supercomputing Institute (Santa Clara, California, May 1987), 45–54.

Голосование X3J13

Символы

Colophon

Camera-ready copy for this book was created by the author (using TEX, LATEX, and TEX macros written by the author), proofed on an Apple LaserWriter II, and printed on a Linotron 300 at Advanced Computer Graphics. The text of the first edition was converted from Scribe format to TEX format by a throwaway program written in Common Lisp. The diagrams in chapter 12 were generated automatically as PostScript code (by a program written in Common Lisp) and integrated into the text by Textures, an implementation of TEX by Blue Sky Research for the Apple Macintosh computer.

The body type is 10-point Times Roman. Chapter titles are in ITC Eras Demi; running heads and chapter subtitles are in ITC Eras Book. The monospace typeface used for program code in both displays and running text is 8.5-point Letter Gothic Bold, somewhat modified by the author through TEX macros for improved legibility. The accent grave (`), accent acute('), circumflex (^), and tilde (~) characters are in 10-point Letter Gothic Bold and adjusted vertically to match the height of the 8.5-point characters. The hyphen (-) was replaced by an en dash (–). The equals sign (=) was replaced by a construction of two em dashes (=), one raised and one lowered, the better to match the other relational characters. The sharp sign (#) is overstruck with two hyphens, one raised and one lowered, to eliminate the vertical gap (#). Special mathematical characters such as square-root signs are in Computer Modern Math. The typefaces used in this book were digitized by Adobe Systems Incorporated, except for Computer Modern Math, which was designed by Donald E. Knuth.