axiom™

The 30 Year Horizon

Manuel Bronstein · *William Burge* · *Timothy Daly*
James Davenport · *Michael Dewar* · *Martin Dunstan*
Albrecht Fortenbacher · *Patrizia Gianni* · *Johannes Grabmeier*
Jocelyn Guidry · *Richard Jenks* · *Larry Lambe*
Michael Monagan · *Scott Morrison* · *William Sit*
Jonathan Steinbach · *Robert Sutor* · *Barry Trager*
Stephen Watt · *Jim Wen* · *Clifton Williamson*

Volume 9: Axiom Compiler

May 26, 2019

1e65374476c066b133978454cb7909ace373d713

Inclusion of names in the list of credits is based on historical information and is as accurate as possible. Inclusion of names does not in any way imply an endorsement but represents historical influence on Axiom development.

Michael Albaugh	Cyril Alberga	Roy Adler
Christian Aistleitner	Richard Anderson	George Andrews
Jerry Archibald	S.J. Atkins	Jeremy Avigad
Knut Bahr	Henry Baker	Martin Baker
Stephen Balzac	Yurij Baransky	David R. Barton
Thomas Baruchel	Gerald Baumgartner	Gilbert Baumslag
Michael Becker	Nelson H. F. Beebe	Jay Belanger
David Bindel	Fred Blair	Vladimir Bondarenko
Mark Botch	Raoul Bourquin	Alexandre Bouyer
Karen Braman	Wolfgang Brehm	Peter A. Broadbery
Martin Brock	Manuel Bronstein	Christopher Brown
Stephen Buchwald	Florian Bundschuh	Luanne Burns
William Burge	Ralph Byers	Quentin Carpent
Jacques Carette	Pierre Casteran	Robert Cavines
Pablo Cayuela	Bruce Char	Ondrej Certik
Tzu-Yi Chen	Bobby Cheng	Cheekai Chin
David V. Chudnovsky	Gregory V. Chudnovsky	Mark Clements
Roland Coeurjoly	Emil Cohen	Hirsh Cohen
Josh Cohen	James Cloos	Jia Zhao Cong
Christophe Conil	Don Coppersmith	George Corliss
Robert Corless	Gary Cornell	Frank Costa
Meino Cramer	Karl Crary	Jeremy Du Croz
David Cyganski	Nathaniel Daly	Timothy Daly Sr.
Timothy Daly Jr.	James H. Davenport	David Day
James Demmel	Didier Deshommes	Michael Dewar
Inderjit Dhillon	Jack Dongarra	Jean Della Dora
Gabriel Dos Reis	Claire DiCrescendo	Sam Dooley
Nicolas James Doye	Zlatko Drmac	Lionel Ducos
Iain Duff	Lee Duhem	Martin Dunstan
Brian Dupee	Dominique Duval	Robert Edwards
Hans-Dieter Ehrich	Heow Eide-Goodman	Carl Engelman
Lars Erickson	Mark Fahey	William Farmer
Richard Fateman	Bertfried Fauser	Stuart Feldman
John Fletcher	Brian Ford	Albrecht Fortenbacher
George Frances	Constantine Frangos	Timothy Freeman
Korrinn Fu	Marc Gaetano	Rudiger Gebauer
Van de Geijn	Kathy Gerber	Patricia Gianni
Gustavo Goertkin	Samantha Goldrich	Max Goldstein
Holger Gollan	Teresa Gomez-Diaz	Ralph Gomory
Laureano Gonzalez-Vega	Stephen Gortler	Johannes Grabmeier
Matt Grayson	Martin Griss	Klaus Ebbe Grue
James Griesmer	Vladimir Grinberg	Oswald Gschnitzer
Ming Gu	Fred Gustavson	Jocelyn Guidry
Gaetan Hache	Steve Hague	Satoshi Hamaguchi
Sven Hammarling	Mike Hansen	Richard Hanson
Richard Harke	Joseph Harry	Bill Hart
Vilya Harvey	Martin Hassner	Arthur S. Hathaway
Dan Hatton	Waldek Hebisch	Karl Hegbloom
Ralf Hemmecke	Tony Hearn	Henderson
Antoine Hersen	Nicholas J. Higham	Lou Hodes
Alan Hoffman	Hoon Hong	Roger House
Gernot Hueber	Pietro Iglio	Joan Jaffe
Alejandro Jakubi	Richard Jenks	Bo Kagstrom
William Kahan	Kyriakos Kalorkoti	Kai Kaminski
Grant Keady	Tom Kelsey	Wilfrid Kendall
Tony Kennedy	David Kincaid	Keshav Kini

Knut Korsvold
Ted Kosan
Paul Kosinski
Igor Kozachenko
Fred Krogh
Klaus Kusche
Bernhard Kutzler
Tim Lahey
Larry Lambe
Kaj Laurson
Charles Lawson
George L. Legendre
Franz Lehner
Frederic Lehobey
Michel Levaud
Howard Levy
J. Lewis
Ren-Cang Li
John Lipson
Rudiger Loos
Craig Lucas
Michael Lucks
Richard Luczak
Camm Maguire
Dave Mainey
Francois Maltey
William Martin
Ursula Martin
Osni Marques
Alasdair McAndrew
Bob McElrath
Michael McGettrick
Bob McNeill
Edi Meier
Ian Meikle
David Mentre
Jonathan Millen
Victor S. Miller
Gerard Milmeister
William Miranker
Mohammed Mobarak
H. Michael Moeller
Michael Monagan
Marc Moreno-Maza
Scott Morrison
Joel Moses
Mark Murray
William Naylor
Patrice Naudin
C. Andrew Neff
John Nelder
Godfrey Nolan
Arthur Norman
Jinzhong Niu
Michael O'Connor
Summat Oemrawsingh
Kostas Oikonomou
Humberto Ortiz-Zuazaga
Julian A. Padget
Bill Page
David Parnas
Norm Pass
Susan Pelzel
Michel Petitot
Didier Pinchon
Ayal Pinkus
Frederick H. Pitts
Frank Pfenning
Jose Alfredo Portes
E. Quintana-Orti
Gregorio Quintana-Orti
Beresford Parlett
A. Petitet
Andre Platzer
Peter Poromaas
Greg Puhak
Claude Quitte
Arthur C. Ralfs
Norman Ramsey
Anatoly Raportirenko
Guilherme Reis
Huan Ren
Albert D. Rich
Michael Richardson
Jason Riedy
Renaud Rioboo
Robert Risch
Jean Rivlin
Nicolas Robidoux
Simon Robinson
Raymond Rogers
Michael Rothstein
Martin Rubey
Jeff Rutter
Philip Santas
David Saunders
Alfred Scheerhorn
William Schelter
Gerhard Schneider
Martin Schoenert
Marshall Schor
Frithjof Schulze
Fritz Schwartz
Steven Segletes
V. Sima
Nick Simicich
William Sit
Elena Smirnova
Jacob Nyffeler Smith
Matthieu Sozeau
Srinivasan Seshan
Ken Stanley
Jonathan Steinbach
Fabio Stumbo
Christine Sundaresan
Klaus Sutner
Robert Sutor
Moss E. Sweedler
Eugene Surowitz
Yong Kiam Tan
Max Tegmark
T. Doug Telford
James Thatcher
Laurent Thery
Balbir Thomas
Mike Thomas
Carol Thompson
Dylan Thurston
Francoise Tisseur
Steve Toleque
Dick Toupin
Raymond Toy
Barry Trager
Hale Trotter
Themos T. Tsikas
Gregory Vanuxem
Kresimir Veselic
Christof Voemel
Bernhard Wall
Paul Wang
Stephen Watt
Andreas Weber
Jaap Weel
Al Weis
Juergen Weiss
M. Weller
Mark Wegman
James Wen
Thorsten Werther
Michael Wester
R. Clint Whaley
James T. Wheeler
John M. Wiley
Berhard Will
Clifton J. Williamson
Stephen Wilson
Shmuel Winograd
Robert Wisbauer
Sandra Wityak
Waldemar Wiwianka
Knut Wolf
Yanyang Xiao
Liu Xiaojun
Clifford Yapp
David Yun
Qian Yun
Vadim Zhytnikov
Richard Zippel
Evelyn Zoernack
Bruno Zuercher
Dan Zwillinger

Contents

New Foreword

On October 1, 2001 Axiom was withdrawn from the market and ended life as a commercial product. On September 3, 2002 Axiom was released under the Modified BSD license, including this document. On August 27, 2003 Axiom was released as free and open source software available for download from the Free Software Foundation's website, Savannah.

Work on Axiom has had the generous support of the Center for Algorithms and Interactive Scientific Computation (CAISS) at City College of New York. Special thanks go to Dr. Gilbert Baumslag for his support of the long term goal.

The online version of this documentation is roughly 1000 pages. In order to make printed versions we've broken it up into three volumes. The first volume is tutorial in nature. The second volume is for programmers. The third volume is reference material. We've also added a fourth volume for developers. All of these changes represent an experiment in print-on-demand delivery of documentation. Time will tell whether the experiment succeeded.

Axiom has been in existence for over thirty years. It is estimated to contain about three hundred man-years of research and has, as of September 3, 2003, 143 people listed in the credits. All of these people have contributed directly or indirectly to making Axiom available. Axiom is being passed to the next generation. I'm looking forward to future milestones.

With that in mind I've introduced the theme of the "30 year horizon". We must invent the tools that support the Computational Mathematician working 30 years from now. How will research be done when every bit of mathematical knowledge is online and instantly available? What happens when we scale Axiom by a factor of 100, giving us 1.1 million domains? How can we integrate theory with code? How will we integrate theorems and proofs of the mathematics with space-time complexity proofs and running code? What visualization tools are needed? How do we support the conceptual structures and semantics of mathematics in effective ways? How do we support results from the sciences? How do we teach the next generation to be effective Computational Mathematicians?

The "30 year horizon" is much nearer than it appears.

Tim Daly
CAISS, City College of New York
November 10, 2003 ((iHy))

Chapter 1

The Axiom Compiler

1.1 Makefile

a : **b** → **T**his book is actually a literate program[Knut92] and contains executable source code. In particular, the Makefile for this book is part of the source of the book and is included below.

Chapter 2

Overview

The Spad language is a mathematically oriented language intended for writing computational mathematics. It derives its logical structure from abstract algebra. It features ideas that are still not available in general purpose programming languages, such as selecting overloaded procedures based on the return type as well as the types of the arguments.

The Spad language is heavily influenced by Barbara Liskov's work. It features encapsulation (aka objects), inheritance, and overloading. It has categories which are defined by the exports. Categories are parameterized functors that take arguments which define their behavior.

More details on the language and its high level concepts is available in the Programmers Guide, Volume 3.

The Spad compiler accepts the Spad language and generates a set of files used by the interpreter, detailed in Volume 5.

The compiler does not produce stand-alone executable code. It assumes that it will run inside the interpreter and that the code it generates will be loaded into the interpreter.

Some of the routines are common to both the compiler and the interpreter. Where this happens we have favored the interpreter volume (Volume 5) as the official source location. In each case we will make reference to that volume and the code in it. Thus, the compiler volume should be considered as an extension of the interpreter document.

This volume will go into painful detail of every aspect of compiling Spad code. We will start by defining the input to, and output from the compiler so we know what we are trying to achieve.

Next we will look at the top level data structures used by the compiler. Unfortunately, the compiler uses a large number of "global variables" to pass information and alter control flow. Some of these are used by many routines and some of these are very local to a small subset or a recursion. We will cover the minor ones as they arise.

Next we examine the Pratt parser idea and the Led and Nud concepts, which is used to drive the low level parsing.

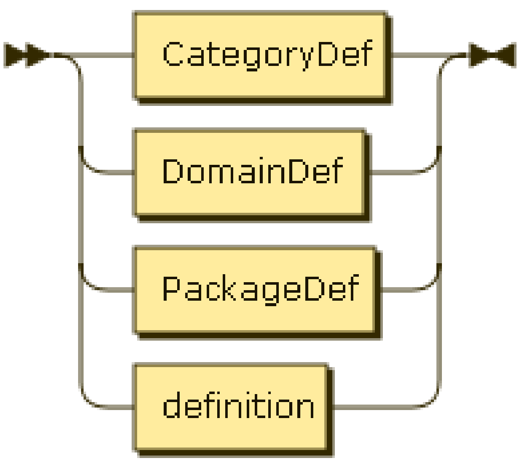
Following that we journey deep into the code, trying our best not to get lost in the details. The code is introduced based on "motivation" rather than in strict execution order or related concept order. We do this to try to make the compiler a "readable novel" rather than a mudmarch through the code. The goal is to keep the reader's interest while trying to be exact. Sometimes this will require detours to discuss subtopics.

"Motivating" a piece of software is a not-very-well established form of narrative writing so we assume your forgiveness if we get it wrong. Worse yet, some of the pieces of the system are "legacy", in that they are no longer used and should be removed. Other parts of the system may have very weak descriptions because we simply do not understand them either. Since this is a living document and the code for the system is actually the code you are reading we will expand parts as we go.

2.1 Syntax by Jacob Smith

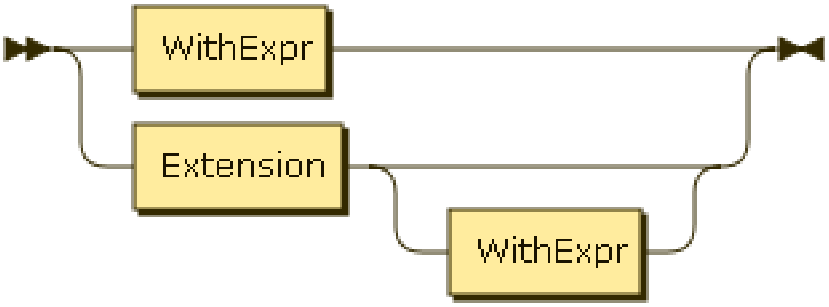
Module

TopLevelDef

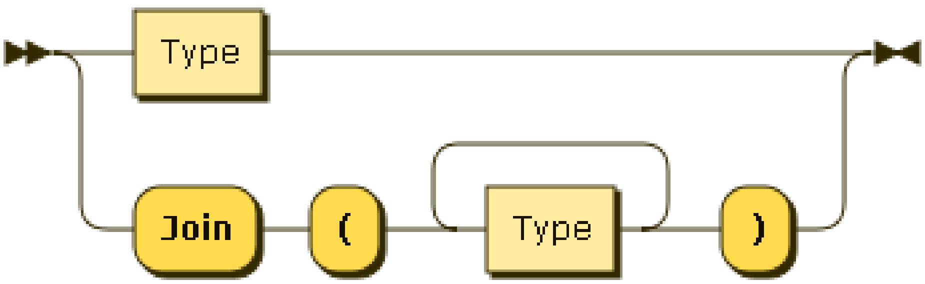


CategoryDef

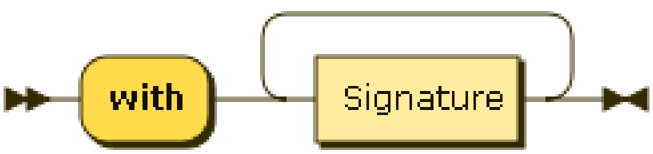
Exports



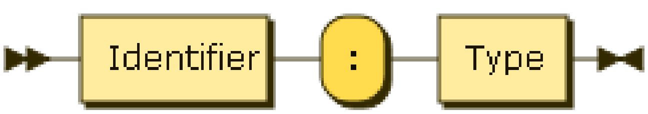
Extension



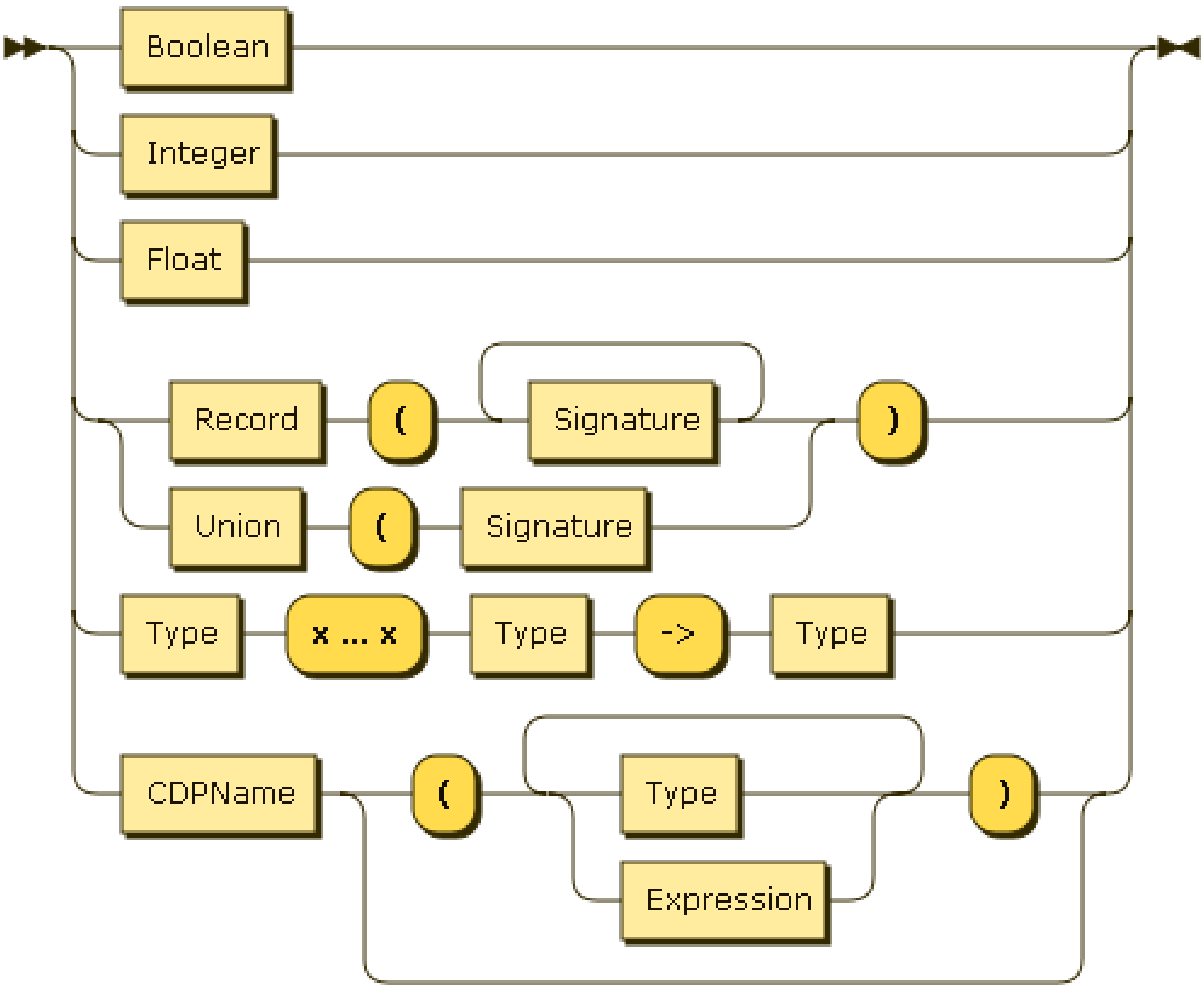
WithExpr



Signature



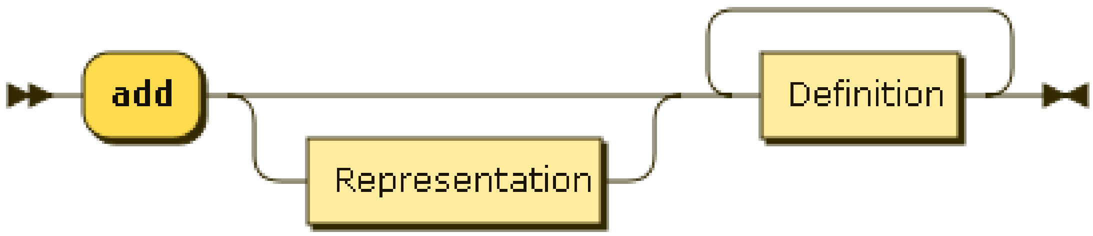
Type



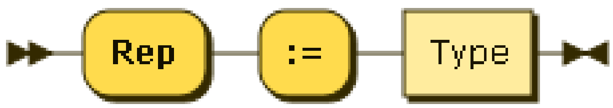
DomainDef

PackageDef

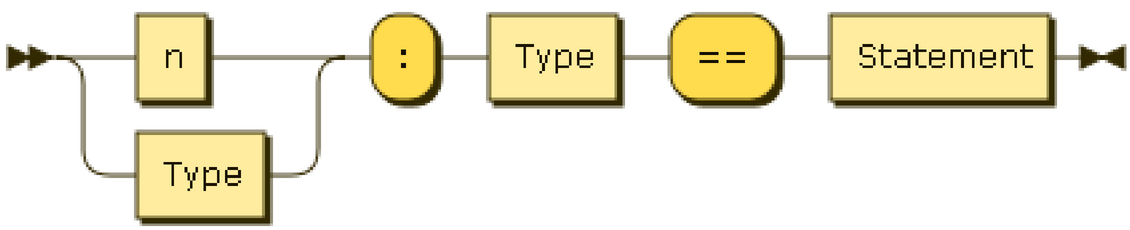
Capsule



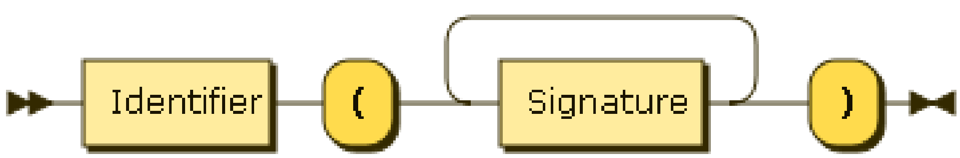
Representation



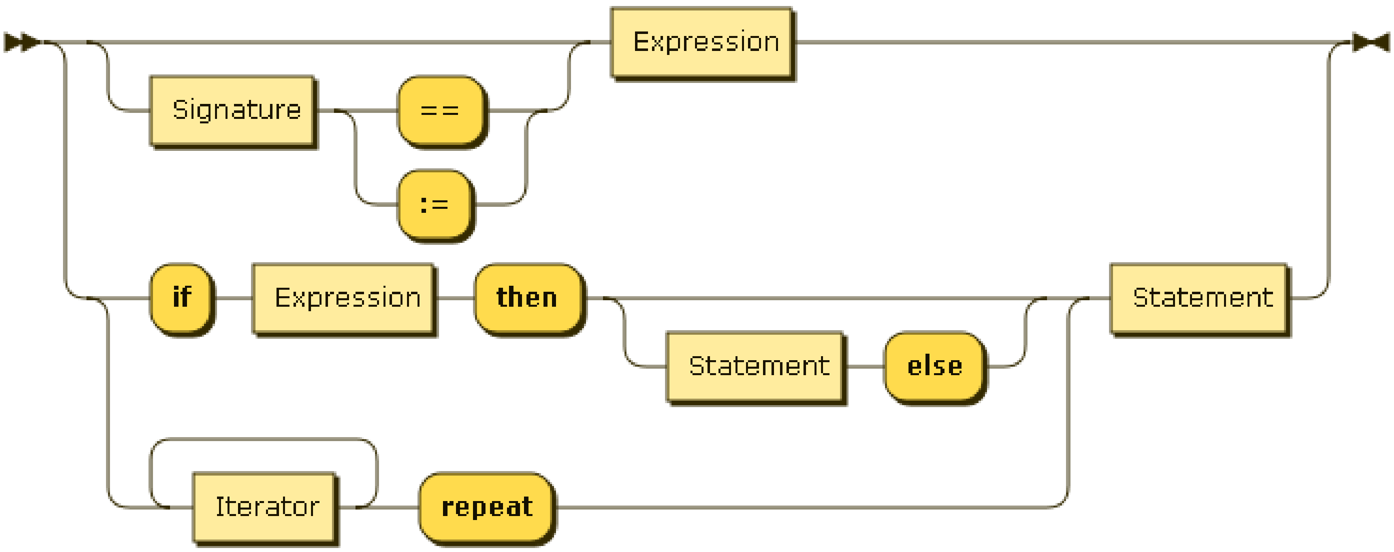
Definition



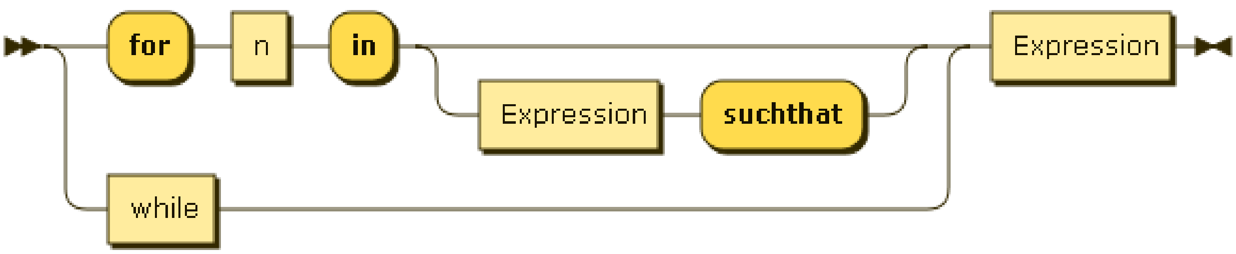
CallForm



Statement



Iterator



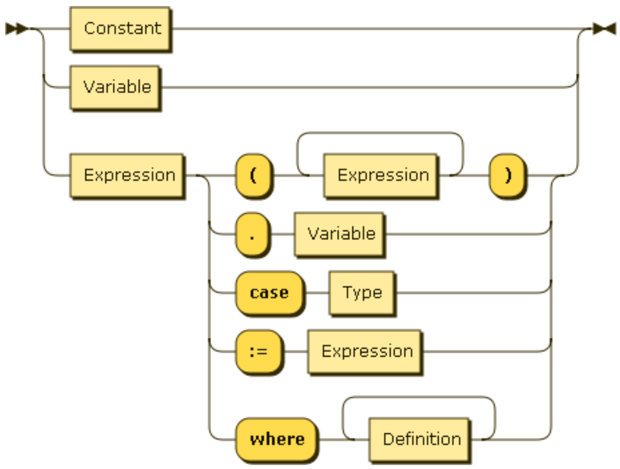
Variable

Constant

Identifier

CDPName

Expression



The Railroad diagrams were generated online [Bott17].

In Smith [Smit10] we find

$$
\begin{array}{rcll}
Module & M & ::= & \Delta^+ \\
TopLevelDef & \Delta & ::= & C \,|\, D \,|\, P \,|\, d \\
CategoryDef & C & ::= & \phi : \text{Category} == E \\
Exports & E & ::= & W \\
 & & | & X\,W^? \\
Extension & X & ::= & \tau \\
 & & | & \text{Join } (\tau^+) \\
WithExpr & W & ::= & \text{with } S^+ \\
Signature & S & ::= & x : \tau \\
Type & \tau & ::= & \text{Boolean} \\
 & & | & \text{Integer} \\
 & & | & \text{Float} \\
 & & | & \text{Record}(S^+) \\
 & & | & \text{Union}(S^+) \\
 & & | & \tau_1 \times \cdots \times \tau_n \to \tau_0 \\
 & & | & \gamma \\
 & & | & \gamma([\tau|e]^+) \\
DomainDef & D & ::= & \phi : E == K \\
PackageDef & P & ::= & \phi : E == K \\
Capsule & K & ::= & \text{add } R^? d^+ \\
Representation & R & ::= & \text{Rep} := \tau \\
Definition & d & ::= & [n|\phi] : \tau == s \\
CallForm & \phi & ::= & x(S^+) \\
Statement & s & ::= & e \\
 & & | & \text{if } e \text{ then } s \\
 & & | & \text{if } e \text{ then } s \text{ else } s \\
 & & | & i^+ \text{ repeat } s \\
 & & | & S == e \\
 & & | & S := e \\
Iterator & i & ::= & \text{for } n \text{ in } e \text{ [suchthat } p]^? \\
 & & | & \text{while } p \\
Expression & e, p & ::= & c \\
 & & | & x^\tau \\
 & & | & e(e^+) \\
 & & | & e.x^\tau \\
 & & | & e \text{ case } \tau \\
 & & | & e := e \\
 & & | & e \text{ where } d^+ \\
Variable & x^\tau & & \\
Constant & c^\tau & & \\
Identifier & x & & \\
CDPName & \gamma & & \text{name of a Spad category, domain, or package}
\end{array}
$$

where $Z^?$ represents an optional Z, Z^+ a non-empty finite sequence of Z, the square brackets are used for grouping.

ToplevelDef A toplevel definition is either a Spad category definition, or a Spad domain definition, or a Spad package definition, or a delayed definition.

CategoryDef A Spad category definition specifies a class of algebras, by declaring the signatures of the required operations. A Spad category definition may extend existing Spad categories with new signatures. For example, the fragment

```
Monoid(): Category == with
   *: (%,%) -> %
   1: %
```

declares `Monoid` as a Spad category with two signatures:

1. the symbol $*$ is a binary operation on the domain belonging to this category
2. the identifier 1 denotes a constant object of a domain belonging to the category being defined.

The following category definition

```
Group(): Category == Monoid with
  inverse: % -> %
```

extends `Monoid` with the `inverse` operation, to capture the mathematical notion of group structure. One can think of Spad categories as *specifying* views on objects.

Exports Definitions for Spad categories, Spad domains, and packages specify exported operations, i.e., the "public interface" in programming languages jargon, through either a *WithExpr*, or an *Extension*, or a combination of both.

WithExpr A *WithExpr* is essentially an unnamed Spad category consisting of a list of operations with signatures (*Signature*).

Extension Definitions for Spad categories and Spad domains may extend existing Spad categories or domains. An extension may specify either a type, or multiple categories through the `Join` operator. The latter form corresponds to multiple inheritance in object-oriented programming languages.

Signature The specification of a tyhpe for an identifer can appear in a *WithExpr*, as a parameter declaration in *CallForm*, as a field of a record or union, or in a local variable definition.

Type A type is a built-in type (`Boolean`, `Integer`, `Float`), a record or union, a function type, the name of a Spad category or Spad domain, or an instantiation of a Spad category or Spad domain. All field names specified by signatures in a record must be distinct. Similarly, all field names specified in a union must be distinct and unique in the enclosing scope; this applies recursively to any other union types directly referenced in the signature list of the union.

`DomainDef` A Spad domain definition provides implemenations for views specified by categories. A domain definition has an interface specification part (*Exports*) stating the categories and possible additional signatures it implements, and an implementation part called *capsule*. The implementation part may define the representation of the object belonging to the domain, and provide definitions for operations declared in its *Exports*. For example, the program fragment

```
IntMonoid(): Monoid == add
  Rep == Integer
  (x:%) * (y:%) == (rep x + rep y)$Integer
  1:% == 0::Integer
```

provides an implementation of `IntMonoid` for the `Monoid` specification as follows:

- the object representation domain is `Integer`;

- "multiplication" of two objects in `IntMonoid` is the value obtained by adding their respective underlying values (returned by the $*$ operator);
- the `Integer` constant 0 is the underlying value of the unit of `IntMonoid`

Note that a Spad domain almost always references the "current domain" using the symbol %.

PackageDef A package definition provides implementations for functions that operate on a Spad domain. Unlike a Spad domain, a package does not define a *Representation* and does not reference the symbol %. Like a Spad domain, it has an *Exports* part and an implementation part.

Capsule The implementation part of a Spad domain or package is its capsule. A capsule mayu specify the representation of a domain (if it si the implementation of a domain), and specifies a sequence of toplevel definitions for operations on the Spad domain objects, or the operators in a package.

sl Representation A Spad domain specifies the underlying representation of its objects by assigning a type expression to the indentifier `Rep`. A *Representation* can occur only in a Spad domain definition.

Definition A (delayed) definition is the binding of an identifier or a function call expression to a Spad category, Spad domain, or an ordinary function. The body of the definition is evaluated when needed. That evaluation may happen only once for a given arugment list. Even though the evaluation is delayed, the body is still fully type checked at the definition point. The Spad language, as understood by the Spad compiler, does not allow ordinary function definitions at the toplevel. However, they are the core of the language understood by the interpreter. For uniformity, we include toplevel function definitions in the Spad subset we describe.

CallForm A call form consists of an identifer and a parenthesized sequence of signatures declaring formal parameters. A call form is needed in the definitions of a Spad category, Spad domain, and function.

Statement Statements appear in the body of function definitions. A statement is either an expression, a one or two-arm if-statment, an iteration where the body of the iteration (a statement) is controlled by a list of iterators, a local variable definition, or an assignment.

Iterator An iterator is either a sequence of items x drawn from a sequence e, possibly filtered by a predicate p, or a repeated evaluation of a predicate.

Expression An expression is either a constant, variable, function call, member selection, type-case expression, an assignment, or a qualifed expression. A qualified expression is an expresstion that contains free variables and is immediately followed by their definitions in a where-clause. We assimilate expressions built with built-in operations – such as addtion on integers, etc. – as function calls.

Variable A variable is the use of a name declared with a given type

Constant A constant is a built-in value, such as $+^{Integer \times Integer \longrightarrow Integer}$, $345^{Integer}$, $true^{Boolean}$, etc.

Identifier An identifier is a finite sequence of characters. The set of identifiers in Spad is countably infinite.

2.1.1 Language features

The Spad programming language supports elements of dependent types, a result of the functorial nature of the data-structuring mechanisms available in Spad. That is, the Spad type system allows types to be function-like objects with arguments that depend on types and values. Dependent types enables an unusually direct style of implementation of mathematical structures.

The Spad programming language also supports general function overloading; in particular, a function can be overloaded on its argument and return types. The overload resolution algorithm exploits all context information, including arguments and target types, to select the best matching function. Implicit conversion is supported through the `coerce` operator.

2.1.2 Sematics

The computational rules used to evaluate Spad programs are those of *eager* semantics (and call-by-value), and the arguments of functions are passed by reference. We sketch the semantics of Spad programs in two ways: small-step operational semantics, and denotational semantics. The small-step operational semantics gives an intuitive idea of the behavior of Spad programs, whereas the denotational semantics lets us associate mathematical functions to Spad programs. The latter allows us to formally talk about the notion of a derivative of a Spad program.

Operational semantics

The Spad language is imperative in the sense that its programs operate on stores by explicit modification. Values of Spad programs can be booleans, integers, floating point numbers, aggregates thereof, or function codes. We denote the collection of values by **Value**, inductively defined as:

Location values:	object locations are in **Value**
Boolean values:	`true` $\in$ **Value** and `false` $\in$ **Value**
Integer Values:	integer constants $n^{Integer}$ are in **Value**
Float values:	float constants f^{Float} are in **Value**
Functions:	If f is a defined function of type $\tau_1 \to \tau_2$
	then the constant $f^{\tau_1 \to \tau_2}$ is in **Value**
Aggregates:	if $c_i^{\tau_i}$ are values of type τ_i in **Value**,
	then the tuple $(c_1^{\tau_1}, \ldots, c_n^{\tau_n})^{\tau_1 \times \cdots \times \tau_n}$ is in **Value**.
	Tuples represent record values.
	Similarly, if c^{τ_1} is in **Value**, so is $c^{\tau_2 \leftarrow \tau_1}$.
	It represents a value of a field of type τ_1 in a union τ_2.

The behavior of a Spad program is a sequence of configurations $\langle p, \sigma, \Gamma \rangle$ where p denotes fragments of Spad constructs, σ the store of values, and Γ the current environment of bindings of variables to types and expressions. The notation $\Gamma, x^\tau == e$ denotes an enviroment obtained by extending Γ with the binding $x^\tau == e$. The $== e$ part may be missing. A store of σ is a mapping from memory locations to Spad values. We use the notation $\sigma[v/l]$ to designate an *updated function* defined by

$$\sigma[v/l] = \begin{cases} v & \text{if } x = l \\ \sigma(x) & \text{otherwise} \end{cases}$$

Each configuration is defined by structural induction on the syntax of Spad.

Denotational semantics

The basic idea of algorithmic differentiation rests on the notion that a computer program computes a function whose range has a ring structure; and the collection of such functions can be endowed with a *differential algebra* structure. The theory of *denotational semantics* is a useful tool. We see a standard denotational semantics of the Spad programming languate that respects the operational semantics outlined above, i.e.

$$t \to^* v^\tau \Rightarrow [[t]] = [[v^\tau]]$$

Variable

$$\langle x^\tau, \sigma, \Gamma \rangle \to \langle \sigma(x^\tau), \sigma, \Gamma \rangle$$

Call-Arguments

$$\frac{\langle e_i, \sigma, \Gamma \rangle \to \langle e_i', \sigma', \Gamma \rangle}{e_0(v_1, \ldots, e_i, \ldots, e_n), \sigma, \Gamma \rangle \to \langle e_0(v_1, \ldots, e_i', \ldots, e_n), \sigma', \Gamma \rangle}$$

where the x_i are parameters of v_0

Call-Operator

$$\frac{\langle e_0, \sigma, \Gamma \rangle \to \langle e_0, \sigma', \Gamma \rangle}{\langle e_0(v_1, \ldots, v_n), \sigma, \Gamma \rangle \to \langle e_0'(v_1, \ldots, v_n), \sigma', \Gamma \rangle}$$

where the x_i are the parameters of v_0

Call

$$\langle v_0^{\tau_0}(v_1^{\tau_1}, \ldots, v_n^{\tau_n}), \sigma, \Gamma \rangle \to \langle v_0^{\tau_0}[v_1^{\tau_1}/x_1, \ldots, v_n^{\tau_n}/x_n], \sigma, \Gamma \rangle$$

where the x_i are parameters of v_0

Qualified Expression

$$\frac{\begin{array}{cc} \langle \delta_1, \sigma, \Gamma_1 \rangle \to \langle \delta_1', \sigma, \Gamma_2 \rangle & \langle \delta_2, \sigma, \Gamma_2 \rangle \to \langle \delta_2', \sigma, \Gamma_3 \rangle \\ \cdots \quad \langle \delta_n, \sigma, \Gamma_n \rangle \to \langle \delta_n', \sigma, \Gamma_{n+1} \rangle & \langle e, \sigma, \Gamma_{n+1} \rangle \to \langle e', \sigma', \Gamma_{n+2} \rangle \end{array}}{\langle e \text{ where } \delta_1 \cdots \delta_n, \sigma, \Gamma_1 \rangle \to \langle e', \sigma', \Gamma_{n+2} \rangle}$$

Sequence-Head

$$\frac{\langle s_1, \sigma_1, \Gamma_1 \rangle \to \langle s_1', \sigma_2, \Gamma_2 \rangle}{\langle s_1; s_2, \sigma_1, \Gamma_1 \rangle \to \langle s_1'; s_2, \sigma_2, \Gamma_2 \rangle}$$

Sequence-Tail

$$\langle v_1; s_2, \sigma, \Gamma \rangle \to \langle s_2, \sigma, \Gamma \rangle$$

If-True

$$\frac{\langle s_1, \sigma, \Gamma \rangle \to \langle s_1', \sigma', \Gamma' \rangle}{\langle \text{ if true then } s_1; s_2, \sigma, \Gamma \rangle \to \langle s_1', \sigma', \Gamma' \rangle}$$

If-False

$$\frac{\langle s_2, \sigma, \Gamma\rangle \rightarrow \langle s_2', \sigma', \Gamma'\rangle}{\langle \text{ if false then } s_1; s_2, \sigma, \Gamma\rangle \rightarrow \langle s_2', \sigma', \Gamma'\rangle}$$

Assignment-Left

$$\frac{\langle e_1, \sigma_0, \Gamma\rangle \rightarrow \langle e_1', \sigma_1, \Gamma\rangle}{\langle e_1 := e_2, \sigma, \Gamma\rangle \rightarrow \langle e_1' := e_2, \sigma_1, \Gamma\rangle}$$

Assignment-Right

$$\frac{\langle e_2, \sigma, \Gamma\rangle \rightarrow \langle e_2', \sigma', \Gamma\rangle}{\langle l := e_2, \sigma, \Gamma\rangle \rightarrow \langle l := e_2', \sigma', \Gamma\rangle}$$

Assignment

$$\frac{}{\langle l := v^\tau, \sigma, \Gamma\rangle \rightarrow \langle v^\tau, \sigma[v^\tau/l], \Gamma\rangle}$$

Immediate Definition

$$\frac{\langle e, \sigma, \Gamma\rangle \rightarrow \langle e', \sigma_1, \Gamma\rangle}{\langle x : \tau := e, \sigma, \Gamma\rangle \rightarrow \langle x : \tau := e', \sigma_1, \Gamma\rangle}$$

Immediate Definition

$$\langle x : \tau := v^\tau, \sigma, \Gamma\rangle \rightarrow \langle v^\tau, \sigma, \Gamma, x^\tau == v^\tau\rangle$$

2.2 The Input

```
)abbrev domain EQ Equation
--FOR THE BENEFIT  OF LIBAXO GENERATION
++ Author: Stephen M. Watt, enhancements by Johannes Grabmeier
++ Date Created: April 1985
++ Date Last Updated: June 3, 1991; September 2, 1992
++ Basic Operations: =
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: equation
++ Examples:
++ References:
++ Description:
++ Equations as mathematical objects.  All properties of the basis domain,
++ e.g. being an abelian group are carried over the equation domain, by
++ performing the structural operations on the left and on the
++ right hand side.
--   The interpreter translates "=" to "equation".  Otherwise, it will
--   find a modemap for "=" in the domain of the arguments.

Equation(S: Type): public == private where
  Ex ==> OutputForm
  public ==> Type with
    "=": (S, S) -> $
```

```
    ++ a=b creates an equation.
equation: (S, S) -> $
    ++ equation(a,b) creates an equation.
swap: $ -> $
    ++ swap(eq) interchanges left and right hand side of equation eq.
lhs: $ -> S
    ++ lhs(eqn) returns the left hand side of equation eqn.
rhs: $ -> S
    ++ rhs(eqn) returns the right hand side of equation eqn.
map: (S -> S, $) -> $
    ++ map(f,eqn) constructs a new equation by applying f to both
    ++ sides of eqn.
if S has InnerEvalable(Symbol,S) then
         InnerEvalable(Symbol,S)
if S has SetCategory then
    SetCategory
    CoercibleTo Boolean
    if S has Evalable(S) then
       eval: ($, $) -> $
        ++ eval(eqn, x=f) replaces x by f in equation eqn.
       eval: ($, List $) -> $
        ++ eval(eqn, [x1=v1, ... xn=vn]) replaces xi by vi in equation eqn.
if S has AbelianSemiGroup then
    AbelianSemiGroup
    "+": (S, $) -> $
        ++ x+eqn produces a new equation by adding x to both sides of
        ++ equation eqn.
    "+": ($, S) -> $
        ++ eqn+x produces a new equation by adding x to  both sides of
        ++ equation eqn.
if S has AbelianGroup then
    AbelianGroup
    leftZero : $ -> $
      ++ leftZero(eq) subtracts the left hand side.
    rightZero : $ -> $
      ++ rightZero(eq) subtracts the right hand side.
    "-": (S, $) -> $
        ++ x-eqn produces a new equation by subtracting both sides of
        ++ equation eqn from x.
    "-": ($, S) -> $
        ++ eqn-x produces a new equation by subtracting x from
        ++ both sides of equation eqn.
if S has SemiGroup then
    SemiGroup
    "*": (S, $) -> $
        ++ x*eqn produces a new equation by multiplying both sides of
        ++ equation eqn by x.
    "*": ($, S) -> $
        ++ eqn*x produces a new equation by multiplying both sides of
        ++ equation eqn by x.
if S has Monoid then
    Monoid
    leftOne : $ -> Union($,"failed")
      ++ leftOne(eq) divides by the left hand side, if possible.
```

```
        rightOne : $ -> Union($,"failed")
          ++ rightOne(eq) divides by the right hand side, if possible.
    if S has Group then
        Group
        leftOne : $ -> Union($,"failed")
          ++ leftOne(eq) divides by the left hand side.
        rightOne : $ -> Union($,"failed")
          ++ rightOne(eq) divides by the right hand side.
    if S has Ring then
      Ring
      BiModule(S,S)
    if S has CommutativeRing then
      Module(S)
      --Algebra(S)
    if S has IntegralDomain then
      factorAndSplit : $ -> List $
        ++ factorAndSplit(eq) make the right hand side 0 and
        ++ factors the new left hand side. Each factor is equated
        ++ to 0 and put into the resulting list without repetitions.
    if S has PartialDifferentialRing(Symbol) then
      PartialDifferentialRing(Symbol)
    if S has Field then
      VectorSpace(S)
      "/": ($, $) -> $
          ++ e1/e2 produces a new equation by dividing the left and right
          ++ hand sides of equations e1 and e2.
      inv: $ -> $
          ++ inv(x) returns the multiplicative inverse of x.
    if S has ExpressionSpace then
        subst: ($, $) -> $
             ++ subst(eq1,eq2) substitutes eq2 into both sides of eq1
             ++ the lhs of eq2 should be a kernel

  private ==> add
    Rep := Record(lhs: S, rhs: S)
    eq1,eq2: $
    s : S
    if S has IntegralDomain then
        factorAndSplit eq ==
          (S has factor : S -> Factored S) =>
            eq0 := rightZero eq
            [equation(rcf.factor,0) for rcf in factors factor lhs eq0]
          [eq]
    l:S = r:S      == [l, r]
    equation(l, r) == [l, r]    -- hack!  See comment above.
    lhs eqn        == eqn.lhs
    rhs eqn        == eqn.rhs
    swap eqn     == [rhs eqn, lhs eqn]
    map(fn, eqn)   == equation(fn(eqn.lhs), fn(eqn.rhs))

    if S has InnerEvalable(Symbol,S) then
        s:Symbol
        ls:List Symbol
        x:S
```

```
        lx:List S
        eval(eqn,s,x) == eval(eqn.lhs,s,x) = eval(eqn.rhs,s,x)
        eval(eqn,ls,lx) == eval(eqn.lhs,ls,lx) = eval(eqn.rhs,ls,lx)
    if S has Evalable(S) then
        eval(eqn1:$, eqn2:$):$ ==
           eval(eqn1.lhs, eqn2 pretend Equation S) =
               eval(eqn1.rhs, eqn2 pretend Equation S)
        eval(eqn1:$, leqn2:List $):$ ==
           eval(eqn1.lhs, leqn2 pretend List Equation S) =
               eval(eqn1.rhs, leqn2 pretend List Equation S)
    if S has SetCategory then
        eq1 = eq2 == (eq1.lhs = eq2.lhs)@Boolean and
                     (eq1.rhs = eq2.rhs)@Boolean
        coerce(eqn:$):Ex == eqn.lhs::Ex = eqn.rhs::Ex
        coerce(eqn:$):Boolean == eqn.lhs = eqn.rhs
    if S has AbelianSemiGroup then
        eq1 + eq2 == eq1.lhs + eq2.lhs = eq1.rhs + eq2.rhs
        s + eq2 == [s,s] + eq2
        eq1 + s == eq1 + [s,s]
    if S has AbelianGroup then
        - eq == (- lhs eq) = (-rhs eq)
        s - eq2 == [s,s] - eq2
        eq1 - s == eq1 - [s,s]
        leftZero eq == 0 = rhs eq - lhs eq
        rightZero eq == lhs eq - rhs eq = 0
        0 == equation(0$S,0$S)
        eq1 - eq2 == eq1.lhs - eq2.lhs = eq1.rhs - eq2.rhs
    if S has SemiGroup then
        eq1:$ * eq2:$ == eq1.lhs * eq2.lhs = eq1.rhs * eq2.rhs
        l:S   * eqn:$ == l       * eqn.lhs = l       * eqn.rhs
        l:S * eqn:$  ==  l * eqn.lhs    =    l * eqn.rhs
        eqn:$ * l:S  ==  eqn.lhs * l    =    eqn.rhs * l
        -- We have to be a bit careful here: raising to a +ve integer is OK
        -- (since it's the equivalent of repeated multiplication)
        -- but other powers may cause contradictions
        -- Watch what else you add here! JHD 2/Aug 1990
    if S has Monoid then
        1 == equation(1$S,1$S)
        recip eq ==
          (lh := recip lhs eq) case "failed" => "failed"
          (rh := recip rhs eq) case "failed" => "failed"
          [lh :: S, rh :: S]
        leftOne eq ==
          (re := recip lhs eq) case "failed" => "failed"
          1 = rhs eq * re
        rightOne eq ==
          (re := recip rhs eq) case "failed" => "failed"
          lhs eq * re = 1
    if S has Group then
        inv eq == [inv lhs eq, inv rhs eq]
        leftOne eq == 1 = rhs eq * inv rhs eq
        rightOne eq == lhs eq * inv rhs eq = 1
    if S has Ring then
        characteristic() == characteristic()$S
```

```
      i:Integer * eq:$ == (i::S) * eq
  if S has IntegralDomain then
      factorAndSplit eq ==
        (S has factor : S -> Factored S) =>
          eq0 := rightZero eq
          [equation(rcf.factor,0) for rcf in factors factor lhs eq0]
        (S has Polynomial Integer) =>
          eq0 := rightZero eq
          MF ==> MultivariateFactorize(Symbol, IndexedExponents Symbol, _
             Integer, Polynomial Integer)
          p : Polynomial Integer := (lhs eq0) pretend Polynomial Integer
          [equation((rcf.factor) pretend S,0) for rcf in factors factor(p)$MF]
        [eq]
  if S has PartialDifferentialRing(Symbol) then
      differentiate(eq:$, sym:Symbol):$ ==
         [differentiate(lhs eq, sym), differentiate(rhs eq, sym)]
  if S has Field then
      dimension() == 2 :: CardinalNumber
      eq1:$ / eq2:$ == eq1.lhs / eq2.lhs = eq1.rhs / eq2.rhs
      inv eq == [inv lhs eq, inv rhs eq]
  if S has ExpressionSpace then
      subst(eq1,eq2) ==
          eq3 := eq2 pretend Equation S
          [subst(lhs eq1,eq3),subst(rhs eq1,eq3)]
```

2.3 The Output, the EQ.nrlib directory

The Spad compiler generates several files in a directory named after the input abbreviation. The input file contains an abbreviation line:

```
)abbrev domain EQ Equation
```

for each category, domain, or package. The abbreviation line has 3 parts.

- one of “category”, “domain”, or “package”
- the abbreviation for this domain (8 Uppercase Characters maximum)
- the name of this domain

Since the abbreviation for the Equation domain is EQ, the compiler will put all of its output into a subdirectory called “EQ.nrlib”. The “nrlib” is a port of a very old VMLisp file format, simulated with directories.

For the EQ input file, the compiler will create the following output files, each of which we will explain in detail below.

```
/research/test/int/algebra/EQ.nrlib:
used 216 available 4992900
drwxr-xr-x    2 root root  4096 2010-12-09 11:20 .
drwxr-xr-x 1259 root root 73728 2010-12-09 11:43 ..
-rw-r--r--    1 root root 19228 2010-12-09 11:20 code.lsp
-rw-r--r--    1 root root 34074 2010-12-09 11:20 code.o
-rw-r--r--    1 root root 13543 2010-12-09 11:20 EQ.fn
-rw-r--r--    1 root root 19228 2010-12-09 11:20 EQ.lsp
```

```
 -rw-r--r--    1 root root 36148 2010-12-09 11:20 index.kaf
 -rw-r--r--    1 root root  6236 2010-12-09 11:20 info
```

2.4 The code.lsp and EQ.lsp files

```
(/VERSIONCHECK 2)

(DEFUN |EQ;factorAndSplit;$L;1| (|eq| $)
  (PROG (|eq0| #:G1403 |rcf| #:G1404)
    (RETURN
      (SEQ (COND
             ((|HasSignature| (QREFELT $ 6)
                  (LIST '|factor|
                        (LIST (LIST '|Factored|
                                    (|devaluate| (QREFELT $ 6)))
                              (|devaluate| (QREFELT $ 6)))))
              (SEQ (LETT |eq0| (SPADCALL |eq| (QREFELT $ 8))
                         |EQ;factorAndSplit;$L;1|)
                   (EXIT (PROGN
                           (LETT #:G1403 NIL |EQ;factorAndSplit;$L;1|)
                           (SEQ (LETT |rcf| NIL
                                      |EQ;factorAndSplit;$L;1|)
                                (LETT #:G1404
                                      (SPADCALL
                                       (SPADCALL
                                        (SPADCALL |eq0| (QREFELT $ 9))
                                        (QREFELT $ 11))
                                       (QREFELT $ 15))
                                      |EQ;factorAndSplit;$L;1|)
                                G190
                                (COND
                                  ((OR (ATOM #:G1404)
                                    (PROGN
                                      (LETT |rcf| (CAR #:G1404)
                                       |EQ;factorAndSplit;$L;1|)
                                      NIL))
                                   (GO G191)))
                                (SEQ (EXIT
                                      (LETT #:G1403
                                       (CONS
                                        (SPADCALL (QCAR |rcf|)
                                         (|spadConstant| $ 16)
                                         (QREFELT $ 17))
                                        #:G1403)
                                       |EQ;factorAndSplit;$L;1|)))
                                (LETT #:G1404 (CDR #:G1404)
                                      |EQ;factorAndSplit;$L;1|)
                                (GO G190) G191
                                (EXIT (NREVERSE0 #:G1403)))))))
             ('T (LIST |eq|)))))))

(PUT (QUOTE |EQ;=;2S$;2|) (QUOTE |SPADreplace|) (QUOTE CONS))
```

```
(DEFUN |EQ;=;2S$;2| (|l| |r| $) (CONS |l| |r|))

(PUT (QUOTE |EQ;equation;2S$;3|) (QUOTE |SPADreplace|) (QUOTE CONS))

(DEFUN |EQ;equation;2S$;3| (|l| |r| $) (CONS |l| |r|))

(PUT (QUOTE |EQ;lhs;$S;4|) (QUOTE |SPADreplace|) (QUOTE QCAR))

(DEFUN |EQ;lhs;$S;4| (|eqn| $) (QCAR |eqn|))

(PUT (QUOTE |EQ;rhs;$S;5|) (QUOTE |SPADreplace|) (QUOTE QCDR))

(DEFUN |EQ;rhs;$S;5| (|eqn| $) (QCDR |eqn|))

(DEFUN |EQ;swap;2$;6| (|eqn| $) (CONS (SPADCALL |eqn| (QREFELT $ 21))
 (SPADCALL |eqn| (QREFELT $ 9))))

(DEFUN |EQ;map;M2$;7| (|fn| |eqn| $)
 (SPADCALL
  (SPADCALL (QCAR |eqn|) |fn|)
  (SPADCALL (QCDR |eqn|) |fn|)
  (QREFELT $ 17)))

(DEFUN |EQ;eval;$SS$;8| (|eqn| |s| |x| $)
 (SPADCALL
  (SPADCALL (QCAR |eqn|) |s| |x| (QREFELT $ 26))
  (SPADCALL (QCDR |eqn|) |s| |x| (QREFELT $ 26))
  (QREFELT $ 20)))

(DEFUN |EQ;eval;$LL$;9| (|eqn| |ls| |lx| $)
 (SPADCALL
  (SPADCALL (QCAR |eqn|) |ls| |lx| (QREFELT $ 30))
  (SPADCALL (QCDR |eqn|) |ls| |lx| (QREFELT $ 30))
  (QREFELT $ 20)))

(DEFUN |EQ;eval;3$;10| (|eqn1| |eqn2| $)
 (SPADCALL
  (SPADCALL (QCAR |eqn1|) |eqn2| (QREFELT $ 33))
  (SPADCALL (QCDR |eqn1|) |eqn2| (QREFELT $ 33))
  (QREFELT $ 20)))

(DEFUN |EQ;eval;$L$;11| (|eqn1| |leqn2| $)
 (SPADCALL
  (SPADCALL (QCAR |eqn1|) |leqn2| (QREFELT $ 36))
  (SPADCALL (QCDR |eqn1|) |leqn2| (QREFELT $ 36))
  (QREFELT $ 20)))

(DEFUN |EQ;=;2$B;12| (|eq1| |eq2| $)
 (COND
  ((SPADCALL (QCAR |eq1|) (QCAR |eq2|) (QREFELT $ 39))
   (SPADCALL (QCDR |eq1|) (QCDR |eq2|) (QREFELT $ 39)))
  ((QUOTE T) (QUOTE NIL))))
```

```
(DEFUN |EQ;coerce;$Of;13| (|eqn| $)
 (SPADCALL
  (SPADCALL (QCAR |eqn|) (QREFELT $ 42))
  (SPADCALL (QCDR |eqn|) (QREFELT $ 42))
  (QREFELT $ 43)))

(DEFUN |EQ;coerce;$B;14| (|eqn| $)
 (SPADCALL (QCAR |eqn|) (QCDR |eqn|) (QREFELT $ 39)))

(DEFUN |EQ;+;3$;15| (|eq1| |eq2| $)
 (SPADCALL
  (SPADCALL (QCAR |eq1|) (QCAR |eq2|) (QREFELT $ 46))
  (SPADCALL (QCDR |eq1|) (QCDR |eq2|) (QREFELT $ 46))
  (QREFELT $ 20)))

(DEFUN |EQ;+;S2$;16| (|s| |eq2| $)
 (SPADCALL (CONS |s| |s|) |eq2| (QREFELT $ 47)))

(DEFUN |EQ;+;$S$;17| (|eq1| |s| $)
 (SPADCALL |eq1| (CONS |s| |s|) (QREFELT $ 47)))

(DEFUN |EQ;-;2$;18| (|eq| $)
 (SPADCALL
  (SPADCALL (SPADCALL |eq| (QREFELT $ 9)) (QREFELT $ 50))
  (SPADCALL (SPADCALL |eq| (QREFELT $ 21)) (QREFELT $ 50))
  (QREFELT $ 20)))

(DEFUN |EQ;-;S2$;19| (|s| |eq2| $)
 (SPADCALL (CONS |s| |s|) |eq2| (QREFELT $ 52)))

(DEFUN |EQ;-;$S$;20| (|eq1| |s| $)
 (SPADCALL |eq1| (CONS |s| |s|) (QREFELT $ 52)))

(DEFUN |EQ;leftZero;2$;21| (|eq| $)
 (SPADCALL
  (|spadConstant| $ 16)
  (SPADCALL
   (SPADCALL |eq| (QREFELT $ 21))
   (SPADCALL |eq| (QREFELT $ 9))
   (QREFELT $ 56))
  (QREFELT $ 20)))

(DEFUN |EQ;rightZero;2$;22| (|eq| $)
 (SPADCALL
  (SPADCALL
   (SPADCALL |eq| (QREFELT $ 9))
   (SPADCALL |eq| (QREFELT $ 21))
   (QREFELT $ 56))
  (|spadConstant| $ 16)
  (QREFELT $ 20)))

(DEFUN |EQ;Zero;$;23| ($)
 (SPADCALL (|spadConstant| $ 16) (|spadConstant| $ 16) (QREFELT $ 17)))
```

```
(DEFUN |EQ;-;3$;24| (|eq1| |eq2| $)
 (SPADCALL
  (SPADCALL (QCAR |eq1|) (QCAR |eq2|) (QREFELT $ 56))
  (SPADCALL (QCDR |eq1|) (QCDR |eq2|) (QREFELT $ 56))
  (QREFELT $ 20)))

(DEFUN |EQ;*;3$;25| (|eq1| |eq2| $)
 (SPADCALL
  (SPADCALL (QCAR |eq1|) (QCAR |eq2|) (QREFELT $ 58))
  (SPADCALL (QCDR |eq1|) (QCDR |eq2|) (QREFELT $ 58))
  (QREFELT $ 20)))

(DEFUN |EQ;*;S2$;26| (|l| |eqn| $)
 (SPADCALL
  (SPADCALL |l| (QCAR |eqn|) (QREFELT $ 58))
  (SPADCALL |l| (QCDR |eqn|) (QREFELT $ 58))
  (QREFELT $ 20)))

(DEFUN |EQ;*;S2$;27| (|l| |eqn| $)
 (SPADCALL
  (SPADCALL |l| (QCAR |eqn|) (QREFELT $ 58))
  (SPADCALL |l| (QCDR |eqn|) (QREFELT $ 58))
  (QREFELT $ 20)))

(DEFUN |EQ;*;$S$;28| (|eqn| |l| $)
 (SPADCALL
  (SPADCALL (QCAR |eqn|) |l| (QREFELT $ 58))
  (SPADCALL (QCDR |eqn|) |l| (QREFELT $ 58))
  (QREFELT $ 20)))

(DEFUN |EQ;One;$;29| ($)
 (SPADCALL (|spadConstant| $ 62) (|spadConstant| $ 62) (QREFELT $ 17)))

(DEFUN |EQ;recip;$U;30| (|eq| $)
 (PROG (|lh| |rh|)
  (RETURN
   (SEQ
    (LETT |lh|
     (SPADCALL (SPADCALL |eq| (QREFELT $ 9)) (QREFELT $ 65))
     |EQ;recip;$U;30|)
    (EXIT
     (COND
      ((QEQCAR |lh| 1) (CONS 1 "failed"))
      ('T
       (SEQ
        (LETT |rh|
         (SPADCALL (SPADCALL |eq| (QREFELT $ 21)) (QREFELT $ 65))
         |EQ;recip;$U;30|)
        (EXIT
         (COND
          ((QEQCAR |rh| 1) (CONS 1 "failed"))
          ('T
            (CONS 0
             (CONS (QCDR |lh|) (QCDR |rh|))))))))))))))
```

```
(DEFUN |EQ;leftOne;$U;31| (|eq| $)
 (PROG (|re|)
  (RETURN
   (SEQ
    (LETT |re|
     (SPADCALL (SPADCALL |eq| (QREFELT $ 9)) (QREFELT $ 65))
     |EQ;leftOne;$U;31|)
    (EXIT
     (COND
      ((QEQCAR |re| 1) (CONS 1 "failed"))
      ('T
       (CONS 0
        (SPADCALL
         (|spadConstant| $ 62)
         (SPADCALL (SPADCALL |eq| (QREFELT $ 21)) (QCDR |re|) (QREFELT $ 58))
         (QREFELT $ 20)))))))))))


(DEFUN |EQ;rightOne;$U;32| (|eq| $)
 (PROG (|re|)
  (RETURN
   (SEQ
    (LETT |re|
     (SPADCALL (SPADCALL |eq| (QREFELT $ 21)) (QREFELT $ 65))
     |EQ;rightOne;$U;32|)
    (EXIT
     (COND
      ((QEQCAR |re| 1) (CONS 1 "failed"))
      ('T
       (CONS 0
        (SPADCALL
         (SPADCALL (SPADCALL |eq| (QREFELT $ 9)) (QCDR |re|) (QREFELT $ 58))
         (|spadConstant| $ 62)
         (QREFELT $ 20)))))))))))


(DEFUN |EQ;inv;2$;33| (|eq| $)
  (CONS (SPADCALL (SPADCALL |eq| (QREFELT $ 9)) (QREFELT $ 69))
        (SPADCALL (SPADCALL |eq| (QREFELT $ 21)) (QREFELT $ 69))))

(DEFUN |EQ;leftOne;$U;34| (|eq| $)
  (CONS 0
        (SPADCALL (|spadConstant| $ 62)
            (SPADCALL (SPADCALL |eq| (QREFELT $ 21))
                (SPADCALL (SPADCALL |eq| (QREFELT $ 21))
                    (QREFELT $ 69))
                (QREFELT $ 58))
            (QREFELT $ 20))))

(DEFUN |EQ;rightOne;$U;35| (|eq| $)
  (CONS 0
        (SPADCALL
```

```
            (SPADCALL (SPADCALL |eq| (QREFELT $ 9))
                (SPADCALL (SPADCALL |eq| (QREFELT $ 21))
                    (QREFELT $ 69))
                (QREFELT $ 58))
            (|spadConstant| $ 62) (QREFELT $ 20))))

(DEFUN |EQ;characteristic;Nni;36| ($) (SPADCALL (QREFELT $ 72)))

(DEFUN |EQ;*;I2$;37| (|i| |eq| $)
 (SPADCALL (SPADCALL |i| (QREFELT $ 75)) |eq| (QREFELT $ 60)))

(DEFUN |EQ;factorAndSplit;$L;38| (|eq| $)
  (PROG (#:G1488 #:G1489 |eq0| |p| #:G1490 |rcf| #:G1491)
    (RETURN
      (SEQ (COND
             ((|HasSignature| (QREFELT $ 6)
                  (LIST '|factor|
                        (LIST (LIST '|Factored|
                                    (|devaluate| (QREFELT $ 6)))
                              (|devaluate| (QREFELT $ 6)))))
              (SEQ (LETT |eq0| (SPADCALL |eq| (QREFELT $ 8))
                         |EQ;factorAndSplit;$L;38|)
                   (EXIT (PROGN
                           (LETT #:G1488 NIL |EQ;factorAndSplit;$L;38|)
                           (SEQ (LETT |rcf| NIL
                                      |EQ;factorAndSplit;$L;38|)
                                (LETT #:G1489
                                      (SPADCALL
                                       (SPADCALL
                                        (SPADCALL |eq0| (QREFELT $ 9))
                                        (QREFELT $ 11))
                                       (QREFELT $ 15))
                                      |EQ;factorAndSplit;$L;38|)
                                G190
                                (COND
                                  ((OR (ATOM #:G1489)
                                    (PROGN
                                      (LETT |rcf| (CAR #:G1489)
                                       |EQ;factorAndSplit;$L;38|)
                                      NIL))
                                   (GO G191)))
                                (SEQ (EXIT
                                      (LETT #:G1488
                                       (CONS
                                        (SPADCALL (QCAR |rcf|)
                                         (|spadConstant| $ 16)
                                         (QREFELT $ 17))
                                        #:G1488)
                                       |EQ;factorAndSplit;$L;38|)))
                                (LETT #:G1489 (CDR #:G1489)
                                      |EQ;factorAndSplit;$L;38|)
                                (GO G190) G191
                                (EXIT (NREVERSE0 #:G1488)))))))
             ((EQUAL (QREFELT $ 6) (|Polynomial| (|Integer|)))
```

```
              (SEQ (LETT |eq0| (SPADCALL |eq| (QREFELT $ 8))
                         |EQ;factorAndSplit;$L;38|)
                   (LETT |p| (SPADCALL |eq0| (QREFELT $ 9))
                         |EQ;factorAndSplit;$L;38|)
                   (EXIT (PROGN
                           (LETT #:G1490 NIL |EQ;factorAndSplit;$L;38|)
                           (SEQ (LETT |rcf| NIL
                                      |EQ;factorAndSplit;$L;38|)
                                (LETT #:G1491
                                      (SPADCALL
                                       (SPADCALL |p| (QREFELT $ 80))
                                       (QREFELT $ 83))
                                      |EQ;factorAndSplit;$L;38|)
                                G190
                                (COND
                                  ((OR (ATOM #:G1491)
                                    (PROGN
                                      (LETT |rcf| (CAR #:G1491)
                                       |EQ;factorAndSplit;$L;38|)
                                      NIL))
                                   (GO G191)))
                                (SEQ (EXIT
                                      (LETT #:G1490
                                       (CONS
                                        (SPADCALL (QCAR |rcf|)
                                         (|spadConstant| $ 16)
                                         (QREFELT $ 17))
                                        #:G1490)
                                       |EQ;factorAndSplit;$L;38|)))
                                (LETT #:G1491 (CDR #:G1491)
                                      |EQ;factorAndSplit;$L;38|)
                                (GO G190) G191
                                (EXIT (NREVERSE0 #:G1490)))))))
             ('T (LIST |eq|)))))))

(DEFUN |EQ;differentiate;$S$;39| (|eq| |sym| $)
  (CONS (SPADCALL (SPADCALL |eq| (QREFELT $ 9)) |sym| (QREFELT $ 84))
        (SPADCALL (SPADCALL |eq| (QREFELT $ 21)) |sym| (QREFELT $ 84))))


(DEFUN |EQ;dimension;Cn;40| ($) (SPADCALL 2 (QREFELT $ 87)))

(DEFUN |EQ;/;3$;41| (|eq1| |eq2| $)
  (SPADCALL (SPADCALL (QCAR |eq1|) (QCAR |eq2|) (QREFELT $ 89))
      (SPADCALL (QCDR |eq1|) (QCDR |eq2|) (QREFELT $ 89))
      (QREFELT $ 20)))

(DEFUN |EQ;inv;2$;42| (|eq| $)
  (CONS (SPADCALL (SPADCALL |eq| (QREFELT $ 9)) (QREFELT $ 69))
        (SPADCALL (SPADCALL |eq| (QREFELT $ 21)) (QREFELT $ 69))))

(DEFUN |EQ;subst;3$;43| (|eq1| |eq2| $)
  (PROG (|eq3|)
    (RETURN
```

```
      (SEQ (LETT |eq3| |eq2| |EQ;subst;3$;43|)
           (EXIT (CONS (SPADCALL (SPADCALL |eq1| (QREFELT $ 9)) |eq3|
                           (QREFELT $ 92))
                       (SPADCALL (SPADCALL |eq1| (QREFELT $ 21)) |eq3|
                           (QREFELT $ 92))))))))

(DEFUN |Equation| (#:G1503)
 (PROG ()
  (RETURN
   (PROG (#:G1504)
    (RETURN
     (COND
      ((LETT #:G1504
        (|lassocShiftWithFunction|
          (LIST (|devaluate| #:G1503))
          (HGET |$ConstructorCache| '|Equation|)
          '|domainEqualList|)
        |Equation|)
       (|CDRwithIncrement| #:G1504))
      ('T
       (UNWIND-PROTECT
        (PROG1 (|Equation;| #:G1503)
         (LETT #:G1504 T |Equation|))
        (COND
         ((NOT #:G1504) (HREM |$ConstructorCache| '|Equation|)))))))))))

(DEFUN |Equation;| (|#1|)
  (PROG (DV$1 |dv$| $ #:G1502 #:G1501 #:G1500 #:G1499 #:G1498 |pv$|)
    (RETURN
      (PROGN
        (LETT DV$1 (|devaluate| |#1|) |Equation|)
        (LETT |dv$| (LIST '|Equation| DV$1) |Equation|)
        (LETT $ (make-array 98) |Equation|)
        (QSETREFV $ 0 |dv$|)
        (QSETREFV $ 3
            (LETT |pv$|
                  (|buildPredVector| 0 0
                      (LIST (|HasCategory| |#1| '(|Field|))
                            (|HasCategory| |#1| '(|SetCategory|))
                            (|HasCategory| |#1| '(|Ring|))
                            (|HasCategory| |#1|
                                '(|PartialDifferentialRing| (|Symbol|)))
                            (OR (|HasCategory| |#1|
                                    '(|PartialDifferentialRing|
                                      (|Symbol|)))
                                (|HasCategory| |#1| '(|Ring|)))
                            (|HasCategory| |#1| '(|Group|))
                            (|HasCategory| |#1|
                                (LIST '|InnerEvalable| '(|Symbol|)
                                      (|devaluate| |#1|)))
                            (AND (|HasCategory| |#1|
                                     (LIST '|Evalable|
                                      (|devaluate| |#1|)))
                                 (|HasCategory| |#1| '(|SetCategory|)))
```

```
          (|HasCategory| |#1| '(|IntegralDomain|))
          (|HasCategory| |#1| '(|ExpressionSpace|))
          (OR (|HasCategory| |#1| '(|Field|))
              (|HasCategory| |#1| '(|Group|)))
          (OR (|HasCategory| |#1| '(|Group|))
              (|HasCategory| |#1| '(|Ring|)))
          (LETT #:G1502
                (|HasCategory| |#1|
                    '(|CommutativeRing|))
                |Equation|)
          (OR #:G1502 (|HasCategory| |#1| '(|Field|))
              (|HasCategory| |#1| '(|Ring|)))
          (OR #:G1502
              (|HasCategory| |#1| '(|Field|)))
          (LETT #:G1501
                (|HasCategory| |#1| '(|Monoid|))
                |Equation|)
          (OR (|HasCategory| |#1| '(|Group|))
              #:G1501)
          (LETT #:G1500
                (|HasCategory| |#1| '(|SemiGroup|))
                |Equation|)
          (OR (|HasCategory| |#1| '(|Group|)) #:G1501
              #:G1500)
          (LETT #:G1499
                (|HasCategory| |#1|
                    '(|AbelianGroup|))
                |Equation|)
          (OR (|HasCategory| |#1|
                  '(|PartialDifferentialRing|
                    (|Symbol|)))
              #:G1499 #:G1502
              (|HasCategory| |#1| '(|Field|))
              (|HasCategory| |#1| '(|Ring|)))
          (OR #:G1499 #:G1501)
          (LETT #:G1498
                (|HasCategory| |#1|
                    '(|AbelianSemiGroup|))
                |Equation|)
          (OR (|HasCategory| |#1|
                  '(|PartialDifferentialRing|
                    (|Symbol|)))
              #:G1499 #:G1498 #:G1502
              (|HasCategory| |#1| '(|Field|))
              (|HasCategory| |#1| '(|Ring|)))
          (OR (|HasCategory| |#1|
                  '(|PartialDifferentialRing|
                    (|Symbol|)))
              #:G1499 #:G1498 #:G1502
              (|HasCategory| |#1| '(|Field|))
              (|HasCategory| |#1| '(|Group|)) #:G1501
              (|HasCategory| |#1| '(|Ring|)) #:G1500
              (|HasCategory| |#1| '(|SetCategory|)))))
|Equation|))
```

```
(|haddProp| |$ConstructorCache| '|Equation| (LIST DV$1)
    (CONS 1 $))
(|stuffDomainSlots| $)
(QSETREFV $ 6 |#1|)
(QSETREFV $ 7 (|Record| (|:| |lhs| |#1|) (|:| |rhs| |#1|)))
(COND
  ((|testBitVector| |pv$| 9)
   (QSETREFV $ 19
       (CONS (|dispatchFunction| |EQ;factorAndSplit;$L;1|) $))))
(COND
  ((|testBitVector| |pv$| 7)
   (PROGN
     (QSETREFV $ 27
         (CONS (|dispatchFunction| |EQ;eval;$SS$;8|) $))
     (QSETREFV $ 31
         (CONS (|dispatchFunction| |EQ;eval;$LL$;9|) $)))))
(COND
  ((|HasCategory| |#1| (LIST '|Evalable| (|devaluate| |#1|)))
   (PROGN
     (QSETREFV $ 34
         (CONS (|dispatchFunction| |EQ;eval;3$;10|) $))
     (QSETREFV $ 37
         (CONS (|dispatchFunction| |EQ;eval;$L$;11|) $)))))
(COND
  ((|testBitVector| |pv$| 2)
   (PROGN
     (QSETREFV $ 40
         (CONS (|dispatchFunction| |EQ;=;2$B;12|) $))
     (QSETREFV $ 44
         (CONS (|dispatchFunction| |EQ;coerce;$Of;13|) $))
     (QSETREFV $ 45
         (CONS (|dispatchFunction| |EQ;coerce;$B;14|) $)))))
(COND
  ((|testBitVector| |pv$| 23)
   (PROGN
     (QSETREFV $ 47 (CONS (|dispatchFunction| |EQ;+;3$;15|) $))
     (QSETREFV $ 48
         (CONS (|dispatchFunction| |EQ;+;S2$;16|) $))
     (QSETREFV $ 49
         (CONS (|dispatchFunction| |EQ;+;$S$;17|) $)))))
(COND
  ((|testBitVector| |pv$| 20)
   (PROGN
     (QSETREFV $ 51 (CONS (|dispatchFunction| |EQ;-;2$;18|) $))
     (QSETREFV $ 53
         (CONS (|dispatchFunction| |EQ;-;S2$;19|) $))
     (QSETREFV $ 54
         (CONS (|dispatchFunction| |EQ;-;$S$;20|) $))
     (QSETREFV $ 57
         (CONS (|dispatchFunction| |EQ;leftZero;2$;21|) $))
     (QSETREFV $ 8
         (CONS (|dispatchFunction| |EQ;rightZero;2$;22|) $))
     (QSETREFV $ 55
         (CONS IDENTITY
```

```
               (FUNCALL (|dispatchFunction| |EQ;Zero;$;23|) $)))
     (QSETREFV $ 52 (CONS (|dispatchFunction| |EQ;-;3$;24|) $)))))
(COND
  ((|testBitVector| |pv$| 18)
   (PROGN
     (QSETREFV $ 59 (CONS (|dispatchFunction| |EQ;*;3$;25|) $))
     (QSETREFV $ 60
         (CONS (|dispatchFunction| |EQ;*;S2$;26|) $))
     (QSETREFV $ 60
         (CONS (|dispatchFunction| |EQ;*;S2$;27|) $))
     (QSETREFV $ 61
         (CONS (|dispatchFunction| |EQ;*;$S$;28|) $)))))
(COND
  ((|testBitVector| |pv$| 16)
   (PROGN
     (QSETREFV $ 63
         (CONS IDENTITY
               (FUNCALL (|dispatchFunction| |EQ;One;$;29|) $)))
     (QSETREFV $ 66
         (CONS (|dispatchFunction| |EQ;recip;$U;30|) $))
     (QSETREFV $ 67
         (CONS (|dispatchFunction| |EQ;leftOne;$U;31|) $))
     (QSETREFV $ 68
         (CONS (|dispatchFunction| |EQ;rightOne;$U;32|) $)))))
(COND
  ((|testBitVector| |pv$| 6)
   (PROGN
     (QSETREFV $ 70
         (CONS (|dispatchFunction| |EQ;inv;2$;33|) $))
     (QSETREFV $ 67
         (CONS (|dispatchFunction| |EQ;leftOne;$U;34|) $))
     (QSETREFV $ 68
         (CONS (|dispatchFunction| |EQ;rightOne;$U;35|) $)))))
(COND
  ((|testBitVector| |pv$| 3)
   (PROGN
     (QSETREFV $ 73
         (CONS (|dispatchFunction| |EQ;characteristic;Nni;36|)
               $))
     (QSETREFV $ 76
         (CONS (|dispatchFunction| |EQ;*;I2$;37|) $)))))
(COND
  ((|testBitVector| |pv$| 9)
   (QSETREFV $ 19
       (CONS (|dispatchFunction| |EQ;factorAndSplit;$L;38|) $))))
(COND
  ((|testBitVector| |pv$| 4)
   (QSETREFV $ 85
       (CONS (|dispatchFunction| |EQ;differentiate;$S$;39|) $))))
(COND
  ((|testBitVector| |pv$| 1)
   (PROGN
     (QSETREFV $ 88
         (CONS (|dispatchFunction| |EQ;dimension;Cn;40|) $))
```

```
          (QSETREFV $ 90 (CONS (|dispatchFunction| |EQ;/;3$;41|) $))
          (QSETREFV $ 70
              (CONS (|dispatchFunction| |EQ;inv;2$;42|) $)))))
     (COND
       ((|testBitVector| |pv$| 10)
        (QSETREFV $ 93
            (CONS (|dispatchFunction| |EQ;subst;3$;43|) $))))
     $))))

(setf (get '|Equation| '|infovec|)
    (LIST '#(NIL NIL NIL NIL NIL NIL (|local| |#1|) '|Rep|
             (0 . |rightZero|) |EQ;lhs;$S;4| (|Factored| $)
             (5 . |factor|)
             (|Record| (|:| |factor| 6) (|:| |exponent| 74))
             (|List| 12) (|Factored| 6) (10 . |factors|) (15 . |Zero|)
             |EQ;equation;2S$;3| (|List| $) (19 . |factorAndSplit|)
             |EQ;=;2S$;2| |EQ;rhs;$S;5| |EQ;swap;2$;6| (|Mapping| 6 6)
             |EQ;map;M2$;7| (|Symbol|) (24 . |eval|) (31 . |eval|)
             (|List| 25) (|List| 6) (38 . |eval|) (45 . |eval|)
             (|Equation| 6) (52 . |eval|) (58 . |eval|) (|List| 32)
             (64 . |eval|) (70 . |eval|) (|Boolean|) (76 . =) (82 . =)
             (|OutputForm|) (88 . |coerce|) (93 . =) (99 . |coerce|)
             (104 . |coerce|) (109 . +) (115 . +) (121 . +) (127 . +)
             (133 . -) (138 . -) (143 . -) (149 . -) (155 . -)
             (161 . |Zero|) (165 . -) (171 . |leftZero|) (176 . *)
             (182 . *) (188 . *) (194 . *) (200 . |One|) (204 . |One|)
             (|Union| $ '"failed") (208 . |recip|) (213 . |recip|)
             (218 . |leftOne|) (223 . |rightOne|) (228 . |inv|)
             (233 . |inv|) (|NonNegativeInteger|)
             (238 . |characteristic|) (242 . |characteristic|)
             (|Integer|) (246 . |coerce|) (251 . *) (|Factored| 78)
             (|Polynomial| 74)
             (|MultivariateFactorize| 25 (|IndexedExponents| 25) 74 78)
             (257 . |factor|)
             (|Record| (|:| |factor| 78) (|:| |exponent| 74))
             (|List| 81) (262 . |factors|) (267 . |differentiate|)
             (273 . |differentiate|) (|CardinalNumber|)
             (279 . |coerce|) (284 . |dimension|) (288 . /) (294 . /)
             (|Equation| $) (300 . |subst|) (306 . |subst|)
             (|PositiveInteger|) (|List| 71) (|SingleInteger|)
             (|String|))
          '#(~= 312 |zero?| 318 |swap| 323 |subtractIfCan| 328 |subst|
             334 |sample| 340 |rightZero| 344 |rightOne| 349 |rhs| 354
             |recip| 359 |one?| 364 |map| 369 |lhs| 375 |leftZero| 380
             |leftOne| 385 |latex| 390 |inv| 395 |hash| 400
             |factorAndSplit| 405 |eval| 410 |equation| 436 |dimension|
             442 |differentiate| 446 |conjugate| 472 |commutator| 478
             |coerce| 484 |characteristic| 499 ^ 503 |Zero| 521 |One|
             525 D 529 = 555 / 567 - 579 + 602 ** 620 * 638)
          '((|unitsKnown| . 12) (|rightUnitary| . 3)
            (|leftUnitary| . 3))
          (CONS (|makeByteWordVec2| 25
                   '(1 15 4 14 5 14 3 5 3 21 21 6 21 17 24 19 25 0 2
                     25 2 7))
```

```
(CONS '#(|VectorSpace&| |Module&|
        |PartialDifferentialRing&| NIL |Ring&| NIL NIL
        NIL NIL |AbelianGroup&| NIL |Group&|
        |AbelianMonoid&| |Monoid&| |AbelianSemiGroup&|
        |SemiGroup&| |SetCategory&| NIL NIL
        |BasicType&| NIL |InnerEvalable&|)
      (CONS '#((|VectorSpace| 6) (|Module| 6)
              (|PartialDifferentialRing| 25)
              (|BiModule| 6 6) (|Ring|)
              (|LeftModule| 6) (|RightModule| 6)
              (|Rng|) (|LeftModule| $$)
              (|AbelianGroup|)
              (|CancellationAbelianMonoid|) (|Group|)
              (|AbelianMonoid|) (|Monoid|)
              (|AbelianSemiGroup|) (|SemiGroup|)
              (|SetCategory|) (|Type|)
              (|CoercibleTo| 41) (|BasicType|)
              (|CoercibleTo| 38)
              (|InnerEvalable| 25 6))
            (|makeByteWordVec2| 97
                '(1 0 0 0 8 1 6 10 0 11 1 14 13 0 15 0
                  6 0 16 1 0 18 0 19 3 6 0 0 25 6 26 3
                  0 0 0 25 6 27 3 6 0 0 28 29 30 3 0 0
                  0 28 29 31 2 6 0 0 32 33 2 0 0 0 0 34
                  2 6 0 0 35 36 2 0 0 0 18 37 2 6 38 0
                  0 39 2 0 38 0 0 40 1 6 41 0 42 2 41 0
                  0 0 43 1 0 41 0 44 1 0 38 0 45 2 6 0
                  0 0 46 2 0 0 0 0 47 2 0 0 6 0 48 2 0
                  0 0 6 49 1 6 0 0 50 1 0 0 0 51 2 0 0
                  0 0 52 2 0 0 6 0 53 2 0 0 0 6 54 0 0
                  0 55 2 6 0 0 0 56 1 0 0 0 57 2 6 0 0
                  0 58 2 0 0 0 0 59 2 0 0 6 0 60 2 0 0
                  0 6 61 0 6 0 62 0 0 0 63 1 6 64 0 65
                  1 0 64 0 66 1 0 64 0 67 1 0 64 0 68 1
                  6 0 0 69 1 0 0 0 70 0 6 71 72 0 0 71
                  73 1 6 0 74 75 2 0 0 74 0 76 1 79 77
                  78 80 1 77 82 0 83 2 6 0 0 25 84 2 0
                  0 0 25 85 1 86 0 71 87 0 0 86 88 2 6
                  0 0 0 89 2 0 0 0 0 90 2 6 0 0 91 92 2
                  0 0 0 0 93 2 2 38 0 0 1 1 20 38 0 1 1
                  0 0 0 22 2 20 64 0 0 1 2 10 0 0 0 93
                  0 22 0 1 1 20 0 0 8 1 16 64 0 68 1 0
                  6 0 21 1 16 64 0 66 1 16 38 0 1 2 0 0
                  23 0 24 1 0 6 0 9 1 20 0 0 57 1 16 64
                  0 67 1 2 97 0 1 1 11 0 0 70 1 2 96 0
                  1 1 9 18 0 19 2 8 0 0 0 34 2 8 0 0 18
                  37 3 7 0 0 25 6 27 3 7 0 0 28 29 31 2
                  0 0 6 6 17 0 1 86 88 2 4 0 0 28 1 2 4
                  0 0 25 85 3 4 0 0 28 95 1 3 4 0 0 25
                  71 1 2 6 0 0 0 1 2 6 0 0 0 1 1 3 0 74
                  1 1 2 41 0 44 1 2 38 0 45 0 3 71 73 2
                  6 0 0 74 1 2 16 0 0 71 1 2 18 0 0 94
                  1 0 20 0 55 0 16 0 63 2 4 0 0 28 1 2
                  4 0 0 25 1 3 4 0 0 28 95 1 3 4 0 0 25
```

```
                                 71 1 2 2 38 0 0 40 2 0 0 6 6 20 2 11
                                 0 0 0 90 2 1 0 0 6 1 1 20 0 0 51 2 20
                                 0 0 0 52 2 20 0 6 0 53 2 20 0 0 6 54
                                 2 23 0 0 0 47 2 23 0 6 0 48 2 23 0 0
                                 6 49 2 6 0 0 74 1 2 16 0 0 71 1 2 18
                                 0 0 94 1 2 20 0 71 0 1 2 20 0 74 0 76
                                 2 23 0 94 0 1 2 18 0 0 0 59 2 18 0 0
                                 6 61 2 18 0 6 0 60)))))
          '|lookupComplete|))
```

2.5 The code.o file

The Spad compiler translates the Spad language into Common Lisp. It eventually invokes the Common Lisp "compile-file" command to output files in binary. Depending on the lisp system this filename can vary (e.g "code.fasl"). The details of how these are used depends on the Common Lisp in use.

By default, Axiom uses Gnu Common Lisp (GCL), which generates ".o" files.

2.6 The info file

```
((* (($ $ $) (|arguments| (|eq2| . $) (|eq1| . $)) (S (* S S S))
     ($ (= $ S S)))
    (($ $ S) (|arguments| (|l| . S) (|eqn| . $)) (S (* S S S))
     ($ (= $ S S)))
    (($ #0=(|Integer|) $) (|arguments| (|i| . #0#) (|eq| . $))
     (S (|coerce| S (|Integer|))) ($ (* $ S $)))
    (($ S $) (|arguments| (|l| . S) (|eqn| . $)) (S (* S S S))
     ($ (= $ S S))))
 (+ (($ $ $) (|arguments| (|eq2| . $) (|eq1| . $)) (S (+ S S S))
     ($ (= $ S S)))
    (($ $ S) (|arguments| (|s| . S) (|eq1| . $)) ($ (+ $ $ $)))
    (($ S $) (|arguments| (|s| . S) (|eq2| . $)) ($ (+ $ $ $))))
 (- (($ $ $) (|arguments| (|eq2| . $) (|eq1| . $)) (S (- S S S))
     ($ (= $ S S)))
    (($ $ S) (|arguments| (|s| . S) (|eq1| . $)) ($ (- $ $ $)))
    (($ $) (|arguments| (|eq| . $)) (S (- S S))
     ($ (|rhs| S $) (|lhs| S $) (= $ S S)))
    (($ S $) (|arguments| (|s| . S) (|eq2| . $)) ($ (- $ $ $))))
 (/ (($ $ $) (|arguments| (|eq2| . $) (|eq1| . $)) (S (/ S S S))
     ($ (= $ S S))))
 (= (($ S S) (|arguments| (|r| . S) (|l| . S)))
    (((|Boolean|) $ $) ((|Boolean|) (|false| (|Boolean|)))
     (|locals| (#:G1393 |Boolean|))
     (|arguments| (|eq2| . $) (|eq1| . $)) (S (= (|Boolean|) S S))))
 (|One| (($) (S (|One| S)) ($ (|equation| $ S S))))
 (|Zero| (($) (S (|Zero| S)) ($ (|equation| $ S S))))
 (|characteristic|
     (((|NonNegativeInteger|))
      (S (|characteristic| (|NonNegativeInteger|)))))
```

```
(|coerce|
    (((|Boolean|) $) (|arguments| (|eqn| . $))
     (S (= (|Boolean|) S S)))
    (((|OutputForm|) $)
     ((|OutputForm|) (= (|OutputForm|) (|OutputForm|) (|OutputForm|)))
     (|arguments| (|eqn| . $)) (S (|coerce| (|OutputForm|) S))))
(|constructor|
    (NIL (|locals|
             (|Rep| |Join| (|SetCategory|)
                    (CATEGORY |domain|
                        (SIGNATURE |construct|
                            ((|Record| (|:| |lhs| S) (|:| |rhs| S)) S
                             S))
                        (SIGNATURE |coerce|
                            ((|OutputForm|)
                             (|Record| (|:| |lhs| S) (|:| |rhs| S))))
                        (SIGNATURE |elt|
                            (S (|Record| (|:| |lhs| S) (|:| |rhs| S))
                               "lhs"))
                        (SIGNATURE |elt|
                            (S (|Record| (|:| |lhs| S) (|:| |rhs| S))
                               "rhs"))
                        (SIGNATURE |setelt|
                            (S (|Record| (|:| |lhs| S) (|:| |rhs| S))
                               "lhs" S))
                        (SIGNATURE |setelt|
                            (S (|Record| (|:| |lhs| S) (|:| |rhs| S))
                               "rhs" S))
                        (SIGNATURE |copy|
                            ((|Record| (|:| |lhs| S) (|:| |rhs| S))
                             (|Record| (|:| |lhs| S) (|:| |rhs| S)))))))))
(|differentiate|
    (($ $ #1=(|Symbol|)) (|arguments| (|sym| . #1#) (|eq| . $))
     (S (|differentiate| S S (|Symbol|))) ($ (|rhs| S $) (|lhs| S $))))
(|dimension|
    ((#2=(|CardinalNumber|))
     (#2# (|coerce| (|CardinalNumber|) (|NonNegativeInteger|)))))
(|equation| (($ S S) (|arguments| (|r| . S) (|l| . S))))
(|eval| (($ $ $) (|arguments| (|eqn2| . $) (|eqn1| . $))
        (S (|eval| S S (|Equation| S))) ($ (= $ S S)))
       (($ $ #3=(|List| $))
        (|arguments| (|leqn2| . #3#) (|eqn1| . $))
        (S (|eval| S S (|List| (|Equation| S)))) ($ (= $ S S)))
       (($ $ #4=(|List| #5=(|Symbol|)) #6=(|List| S))
        (|arguments| (|lx| . #6#) (|ls| . #4#) (|eqn| . $))
        (S (|eval| S S (|List| (|Symbol|)) (|List| S)))
        ($ (= $ S S)))
       (($ $ #5# S) (|arguments| (|x| . S) (|s| . #5#) (|eqn| . $))
        (S (|eval| S S (|Symbol|) S)) ($ (= $ S S))))
(|factorAndSplit|
    (((|List| $) $)
     ((|MultivariateFactorize| (|Symbol|)
          (|IndexedExponents| (|Symbol|)) (|Integer|)
          (|Polynomial| (|Integer|)))
```

```
      (|factor| (|Factored| (|Polynomial| (|Integer|)))
          (|Polynomial| (|Integer|))))
     ((|Factored| S)
      (|factors|
          (|List| (|Record| (|:| |factor| S)
                      (|:| |exponent| (|Integer|))))
          (|Factored| S)))
     ((|Factored| (|Polynomial| (|Integer|)))
      (|factors|
          (|List| (|Record| (|:| |factor| (|Polynomial| (|Integer|)))
                      (|:| |exponent| (|Integer|))))
          (|Factored| (|Polynomial| (|Integer|)))))
     (|locals| (|p| |Polynomial| (|Integer|)) (|eq0| . $))
     (|arguments| (|eq| . $))
     (S (|factor| (|Factored| S) S) (|Zero| S))
     ($ (|rightZero| $ $) (|lhs| S $) (|equation| $ S S))))
 (|inv| (($ $) (|arguments| (|eq| . $)) (S (|inv| S S))
        ($ (|rhs| S $) (|lhs| S $))))
 (|leftOne|
    (((|Union| $ "failed") $) (|locals| (|re| |Union| S "failed"))
     (|arguments| (|eq| . $))
     (S (|recip| (|Union| S "failed") S) (|inv| S S) (|One| S)
        (* S S S))
     ($ (|rhs| S $) (|lhs| S $) (|One| $) (= $ S S))))
 (|leftZero|
    (($ $) (|arguments| (|eq| . $)) (S (|Zero| S) (- S S S))
     ($ (|rhs| S $) (|lhs| S $) (|Zero| $) (= $ S S))))
 (|lhs| ((S $) (|arguments| (|eqn| . $))))
 (|map| (($ #7=(|Mapping| S S) $)
        (|arguments| (|fn| . #7#) (|eqn| . $)) ($ (|equation| $ S S))))
 (|recip| (((|Union| $ "failed") $)
          (|locals| (|rh| |Union| S "failed")
              (|lh| |Union| S "failed"))
          (|arguments| (|eq| . $))
          (S (|recip| (|Union| S "failed") S))
          ($ (|rhs| S $) (|lhs| S $))))
 (|rhs| ((S $) (|arguments| (|eqn| . $))))
 (|rightOne|
    (((|Union| $ "failed") $) (|locals| (|re| |Union| S "failed"))
     (|arguments| (|eq| . $))
     (S (|recip| (|Union| S "failed") S) (|inv| S S) (|One| S)
        (* S S S))
     ($ (|rhs| S $) (|lhs| S $) (= $ S S))))
 (|rightZero|
    (($ $) (|arguments| (|eq| . $)) (S (|Zero| S) (- S S S))
     ($ (|rhs| S $) (|lhs| S $) (= $ S S))))
 (|subst| (($ $ $) (|locals| (|eq3| |Equation| S))
          (|arguments| (|eq2| . $) (|eq1| . $))
          (S (|subst| S S (|Equation| S)))
          ($ (|rhs| S $) (|lhs| S $))))
 (|swap| (($ $) (|arguments| (|eqn| . $)) ($ (|rhs| S $) (|lhs| S $)))))
```

2.7 The EQ.fn file

```
(in-package 'compiler)(init-fn)
(ADD-FN-DATA '(
#S(FN NAME BOOT::|EQ;*;S2$;26| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;rightOne;$U;32| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES
      (BOOT::|spadConstant| VMLISP:QCDR CONS VMLISP:QCAR EQL
          BOOT::QEQCAR COND VMLISP:EXIT CDR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL BOOT::LETT VMLISP:SEQ RETURN)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (BOOT::|spadConstant| VMLISP:QCDR VMLISP:QCAR BOOT::QEQCAR COND
          VMLISP:EXIT VMLISP:QREFELT BOOT:SPADCALL BOOT::LETT
          VMLISP:SEQ RETURN))
#S(FN NAME BOOT::|EQ;lhs;$S;4| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CAR VMLISP:QCAR) RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT
      NIL MACROS (VMLISP:QCAR))
#S(FN NAME BOOT::|EQ;+;3$;15| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;dimension;Cn;40| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;rightZero;2$;22| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES
      (BOOT::|spadConstant| CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (BOOT::|spadConstant| VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;coerce;$Of;13| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;One;$;29| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (CDR BOOT::|spadConstant| CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T) NO-EMIT NIL MACROS
      (BOOT::|spadConstant| VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;inv;2$;42| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL CONS)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;-;$S$;20| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CDR CONS CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
```

```
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;=;2$B;12| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL COND)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL COND))
#S(FN NAME BOOT::|EQ;/;3$;41| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;recip;$U;30| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR LIST* CONS VMLISP:QCAR EQL BOOT::QEQCAR COND
          VMLISP:EXIT CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL
          BOOT::LETT VMLISP:SEQ RETURN)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR BOOT::QEQCAR COND VMLISP:EXIT
          VMLISP:QREFELT BOOT:SPADCALL BOOT::LETT VMLISP:SEQ RETURN))
#S(FN NAME BOOT::|EQ;-;3$;24| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;eval;$L$;11| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;leftZero;2$;21| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES
      (CDR BOOT::|spadConstant| CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (BOOT::|spadConstant| VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;*;S2$;27| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;*;I2$;37| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL) RETURN-TYPE
      NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;eval;3$;10| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
```

```
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;eval;$SS$;8| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;factorAndSplit;$L;38| DEF DEFUN VALUE-TYPE T
      FUN-VALUES NIL CALLEES
      (BOOT:|Integer| BOOT:|Polynomial| EQUAL BOOT:NREVERSE0
          BOOT::|spadConstant| VMLISP:QCAR CONS ATOM VMLISP:EXIT CDR
          CAR BOOT:SPADCALL BOOT::LETT BOOT::|devaluate| LIST SVREF
          VMLISP:QREFELT BOOT::|HasSignature| COND VMLISP:SEQ RETURN)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (BOOT::|spadConstant| VMLISP:QCAR VMLISP:EXIT BOOT:SPADCALL
          BOOT::LETT VMLISP:QREFELT COND VMLISP:SEQ RETURN))
#S(FN NAME BOOT::|EQ;differentiate;$S$;39| DEF DEFUN VALUE-TYPE T
      FUN-VALUES NIL CALLEES
      (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL CONS) RETURN-TYPE NIL
      ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;eval;$LL$;9| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;leftOne;$U;34| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES
      (CDR BOOT::|spadConstant| CAR SVREF VMLISP:QREFELT BOOT:SPADCALL
           CONS)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (BOOT::|spadConstant| VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;map;M2$;7| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
          BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;-;S2$;19| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CDR CONS CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;equation;2S$;3| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES (CONS) RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL
      MACROS NIL)
#S(FN NAME BOOT::|EQ;+;$S$;17| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CDR CONS CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;factorAndSplit;$L;1| DEF DEFUN VALUE-TYPE T
      FUN-VALUES NIL CALLEES
      (BOOT:NREVERSE0 BOOT::|spadConstant| VMLISP:QCAR CONS ATOM
          VMLISP:EXIT CDR CAR BOOT:SPADCALL BOOT::LETT
```

```
        BOOT::|devaluate| LIST SVREF VMLISP:QREFELT
        BOOT::|HasSignature| COND VMLISP:SEQ RETURN)
    RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
    (BOOT::|spadConstant| VMLISP:QCAR VMLISP:EXIT BOOT:SPADCALL
        BOOT::LETT VMLISP:QREFELT COND VMLISP:SEQ RETURN))
#S(FN NAME BOOT::|EQ;*;3$;25| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES
    (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
        BOOT:SPADCALL)
    RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
    (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;Zero;$;23| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES
    (CDR BOOT::|spadConstant| CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
    RETURN-TYPE NIL ARG-TYPES (T) NO-EMIT NIL MACROS
    (BOOT::|spadConstant| VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;characteristic;Nni;36| DEF DEFUN VALUE-TYPE T
    FUN-VALUES NIL CALLEES
    (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL) RETURN-TYPE NIL
    ARG-TYPES (T) NO-EMIT NIL MACROS (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;leftOne;$U;31| DEF DEFUN VALUE-TYPE T FUN-VALUES
    NIL CALLEES
    (VMLISP:QCDR BOOT::|spadConstant| CONS VMLISP:QCAR EQL
        BOOT::QEQCAR COND VMLISP:EXIT CDR CAR SVREF VMLISP:QREFELT
        BOOT:SPADCALL BOOT::LETT VMLISP:SEQ RETURN)
    RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
    (VMLISP:QCDR BOOT::|spadConstant| VMLISP:QCAR BOOT::QEQCAR COND
        VMLISP:EXIT VMLISP:QREFELT BOOT:SPADCALL BOOT::LETT
        VMLISP:SEQ RETURN))
#S(FN NAME BOOT::|EQ;swap;2$;6| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL CONS)
    RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
    (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;-;2$;18| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL) RETURN-TYPE
    NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
    (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;subst;3$;43| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES
    (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL CONS VMLISP:EXIT
         BOOT::LETT VMLISP:SEQ RETURN)
    RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
    (VMLISP:QREFELT BOOT:SPADCALL VMLISP:EXIT BOOT::LETT VMLISP:SEQ
        RETURN))
#S(FN NAME BOOT::|EQ;=;2S$;2| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES (CONS) RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL
    MACROS NIL)
#S(FN NAME BOOT::|EQ;*;$S$;28| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
    CALLEES
    (VMLISP:QCDR CDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
        BOOT:SPADCALL)
    RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
    (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;+;S2$;16| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
```

```
      CALLEES (CDR CONS CAR SVREF VMLISP:QREFELT BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|Equation;| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES
      (BOOT::|EQ;One;$;29| BOOT::|EQ;Zero;$;23|
          BOOT::|dispatchFunction| BOOT::|testBitVector| COND
          BOOT::|Record0| BOOT::|Record| BOOT::|stuffDomainSlots| CONS
          BOOT::|haddProp| BOOT::|HasCategory| BOOT::|buildPredVector|
          SYSTEM:SVSET SETF VMLISP:QSETREFV LIST
          BOOT::|devaluate| BOOT::LETT RETURN)
      RETURN-TYPE NIL ARG-TYPES (T) NO-EMIT NIL MACROS
      (BOOT::|dispatchFunction| COND BOOT::|Record| SETF
          VMLISP:QSETREFV BOOT::LETT RETURN))
#S(FN NAME BOOT::|EQ;coerce;$B;14| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES
      (CDR VMLISP:QCDR VMLISP:QCAR CAR SVREF VMLISP:QREFELT
           BOOT:SPADCALL)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (VMLISP:QCDR VMLISP:QCAR VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;rhs;$S;5| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CDR VMLISP:QCDR) RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT
      NIL MACROS (VMLISP:QCDR))
#S(FN NAME OTHER-FORM DEF NIL VALUE-TYPE NIL FUN-VALUES NIL CALLEES NIL
      RETURN-TYPE NIL ARG-TYPES NIL NO-EMIT NIL MACROS NIL)
#S(FN NAME BOOT::|EQ;inv;2$;33| DEF DEFUN VALUE-TYPE T FUN-VALUES NIL
      CALLEES (CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL CONS)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|EQ;rightOne;$U;35| DEF DEFUN VALUE-TYPE T FUN-VALUES
      NIL CALLEES
      (BOOT::|spadConstant| CDR CAR SVREF VMLISP:QREFELT BOOT:SPADCALL
          CONS)
      RETURN-TYPE NIL ARG-TYPES (T T) NO-EMIT NIL MACROS
      (BOOT::|spadConstant| VMLISP:QREFELT BOOT:SPADCALL))
#S(FN NAME BOOT::|Equation| DEF DEFUN VALUE-TYPE T FUN-VALUES
      (SINGLE-VALUE) CALLEES
      (REMHASH VMLISP:HREM BOOT::|Equation;| PROG1
               BOOT::|CDRwithIncrement| GETHASH VMLISP:HGET
               BOOT::|devaluate| LIST BOOT::|lassocShiftWithFunction|
               BOOT::LETT COND RETURN)
      RETURN-TYPE NIL ARG-TYPES (T) NO-EMIT NIL MACROS
      (VMLISP:HREM PROG1 VMLISP:HGET BOOT::LETT COND RETURN)) ))
```

2.8 The index.kaf file

Each constructor (e.g. EQ) had one library directory (e.g. EQ.nrlib). This directory contained a random access file called the index.kaf file. These files contain runtime information such as the operationAlist and the ConstructorModemap. At system build time we merge all of these .nrlib/index.kaf files into one database, INTERP.daase. Requests to get information from this database are cached so that multiple references do not cause additional disk i/o.

Before getting into the contents, we need to understand the format of an index.kaf file. The kaf file is a random access file, originally used as a database. In the current system we make a pass to combine these files at build time to construct the various daase files.

This is just a file of lisp objects, one after another, in (read) format.

A kaf file starts with an integer, in this case, 35695. This integer gives the byte offset to the index. Due to the way the file is constructed, the index is at the end of the file. To read a kaf file, first read the integer, then seek to that location in the file, and do a (read). This will return the index, in this case:

```
(("slot1Info" 0 32444)
 ("documentation" 0 29640)
 ("ancestors" 0 28691)
 ("parents" 0 28077)
 ("abbreviation" 0 28074)
 ("predicates" 0 25442)
 ("attributes" 0 25304)
 ("signaturesAndLocals" 0 23933)
 ("superDomain" 0 NIL)
 ("operationAlist" 0 20053)
 ("modemaps" 0 17216)
 ("sourceFile" 0 17179)
 ("constructorCategory" 0 15220)
 ("constructorModemap" 0 13215)
 ("constructorKind" 0 13206)
 ("constructorForm" 0 13191)
 ("compilerInfo" 0 4433)
 ("loadTimeStuff" 0 20))
```

This is a list of triples. The first item in each triple is a string that is used as a lookup key (e.g. "operationAlist"). The second element is no longer used. The third element is the byte offset from the beginning of the file.

So to read the "operationAlist" from this file you would:

1. open the index.kaf file
2. (read) the integer
3. (seek) to the integer offset from the beginning of the file
4. (read) the index of triples
5. find the keyword (e.g. "operationAlist") triple
6. select the third element, an integer
7. (seek) to the integer offset from the beginning of the file
8. (read) the "operationAlist"

Note that the information below has been reformatted to fit this document. In order to save space the index.kaf file is does not use prettyprint since it is normally only read by machine.

2.8.1 The index offset byte

```
35695
```

2.8.2 The “loadTimeStuff”

```
(setf (get '|Equation| '|infovec|)
    (LIST '#(NIL NIL NIL NIL NIL NIL (|local| |#1|) '|Rep|
             (0 . |rightZero|) |EQ;lhs;$S;4| (|Factored| $)
             (5 . |factor|)
             (|Record| (|:| |factor| 6) (|:| |exponent| 74))
             (|List| 12) (|Factored| 6) (10 . |factors|) (15 . |Zero|)
             |EQ;equation;2S$;3| (|List| $) (19 . |factorAndSplit|)
             |EQ;=;2S$;2| |EQ;rhs;$S;5| |EQ;swap;2$;6| (|Mapping| 6 6)
             |EQ;map;M2$;7| (|Symbol|) (24 . |eval|) (31 . |eval|)
             (|List| 25) (|List| 6) (38 . |eval|) (45 . |eval|)
             (|Equation| 6) (52 . |eval|) (58 . |eval|) (|List| 32)
             (64 . |eval|) (70 . |eval|) (|Boolean|) (76 . =) (82 . =)
             (|OutputForm|) (88 . |coerce|) (93 . =) (99 . |coerce|)
             (104 . |coerce|) (109 . +) (115 . +) (121 . +) (127 . +)
             (133 . -) (138 . -) (143 . -) (149 . -) (155 . -)
             (161 . |Zero|) (165 . -) (171 . |leftZero|) (176 . *)
             (182 . *) (188 . *) (194 . *) (200 . |One|) (204 . |One|)
             (|Union| $ '"failed") (208 . |recip|) (213 . |recip|)
             (218 . |leftOne|) (223 . |rightOne|) (228 . |inv|)
             (233 . |inv|) (|NonNegativeInteger|)
             (238 . |characteristic|) (242 . |characteristic|)
             (|Integer|) (246 . |coerce|) (251 . *) (|Factored| 78)
             (|Polynomial| 74)
             (|MultivariateFactorize| 25 (|IndexedExponents| 25) 74 78)
             (257 . |factor|)
             (|Record| (|:| |factor| 78) (|:| |exponent| 74))
             (|List| 81) (262 . |factors|) (267 . |differentiate|)
             (273 . |differentiate|) (|CardinalNumber|)
             (279 . |coerce|) (284 . |dimension|) (288 . /) (294 . /)
             (|Equation| $) (300 . |subst|) (306 . |subst|)
             (|PositiveInteger|) (|List| 71) (|SingleInteger|)
             (|String|))
          '#(~= 312 |zero?| 318 |swap| 323 |subtractIfCan| 328 |subst|
             334 |sample| 340 |rightZero| 344 |rightOne| 349 |rhs| 354
             |recip| 359 |one?| 364 |map| 369 |lhs| 375 |leftZero| 380
             |leftOne| 385 |latex| 390 |inv| 395 |hash| 400
             |factorAndSplit| 405 |eval| 410 |equation| 436 |dimension|
             442 |differentiate| 446 |conjugate| 472 |commutator| 478
             |coerce| 484 |characteristic| 499 ^ 503 |Zero| 521 |One|
             525 D 529 = 555 / 567 - 579 + 602 ** 620 * 638)
          '((|unitsKnown| . 12) (|rightUnitary| . 3)
            (|leftUnitary| . 3))
          (CONS (|makeByteWordVec2| 25
                  '(1 15 4 14 5 14 3 5 3 21 21 6 21 17 24 19 25 0 2
                    25 2 7))
                (CONS '#(|VectorSpace&| |Module&|
                        |PartialDifferentialRing&| NIL |Ring&| NIL NIL
                        NIL NIL |AbelianGroup&| NIL |Group&|
                        |AbelianMonoid&| |Monoid&| |AbelianSemiGroup&|
                        |SemiGroup&| |SetCategory&| NIL NIL
                        |BasicType&| NIL |InnerEvalable&|)
                  (CONS '#((|VectorSpace| 6) (|Module| 6)
```

```
   (|PartialDifferentialRing| 25)
   (|BiModule| 6 6) (|Ring|)
   (|LeftModule| 6) (|RightModule| 6)
   (|Rng|) (|LeftModule| $$)
   (|AbelianGroup|)
   (|CancellationAbelianMonoid|) (|Group|)
   (|AbelianMonoid|) (|Monoid|)
   (|AbelianSemiGroup|) (|SemiGroup|)
   (|SetCategory|) (|Type|)
   (|CoercibleTo| 41) (|BasicType|)
   (|CoercibleTo| 38)
   (|InnerEvalable| 25 6))
(|makeByteWordVec2| 97
    '(1 0 0 0 8 1 6 10 0 11 1 14 13 0 15 0
      6 0 16 1 0 18 0 19 3 6 0 0 25 6 26 3
      0 0 0 25 6 27 3 6 0 0 28 29 30 3 0 0
      0 28 29 31 2 6 0 0 32 33 2 0 0 0 0 34
      2 6 0 0 35 36 2 0 0 0 18 37 2 6 38 0
      0 39 2 0 38 0 0 40 1 6 41 0 42 2 41 0
      0 0 43 1 0 41 0 44 1 0 38 0 45 2 6 0
      0 0 46 2 0 0 0 0 47 2 0 0 6 0 48 2 0
      0 0 6 49 1 6 0 0 50 1 0 0 0 51 2 0 0
      0 0 52 2 0 0 6 0 53 2 0 0 0 6 54 0 0
      0 55 2 6 0 0 0 56 1 0 0 0 57 2 6 0 0
      0 58 2 0 0 0 0 59 2 0 0 6 0 60 2 0 0
      0 6 61 0 6 0 62 0 0 0 63 1 6 64 0 65
      1 0 64 0 66 1 0 64 0 67 1 0 64 0 68 1
      6 0 0 69 1 0 0 0 70 0 6 71 72 0 0 71
      73 1 6 0 74 75 2 0 0 74 0 76 1 79 77
      78 80 1 77 82 0 83 2 6 0 0 25 84 2 0
      0 0 25 85 1 86 0 71 87 0 0 86 88 2 6
      0 0 0 89 2 0 0 0 0 90 2 6 0 0 91 92 2
      0 0 0 0 93 2 2 38 0 0 1 1 20 38 0 1 1
      0 0 0 22 2 20 64 0 0 1 2 10 0 0 0 93
      0 22 0 1 1 20 0 0 8 1 16 64 0 68 1 0
      6 0 21 1 16 64 0 66 1 16 38 0 1 2 0 0
      23 0 24 1 0 6 0 9 1 20 0 0 57 1 16 64
      0 67 1 2 97 0 1 1 11 0 0 70 1 2 96 0
      1 1 9 18 0 19 2 8 0 0 0 34 2 8 0 0 18
      37 3 7 0 0 25 6 27 3 7 0 0 28 29 31 2
      0 0 6 6 17 0 1 86 88 2 4 0 0 28 1 2 4
      0 0 25 85 3 4 0 0 28 95 1 3 4 0 0 25
      71 1 2 6 0 0 0 1 2 6 0 0 0 1 1 3 0 74
      1 1 2 41 0 44 1 2 38 0 45 0 3 71 73 2
      6 0 0 74 1 2 16 0 0 71 1 2 18 0 0 94
      1 0 20 0 55 0 16 0 63 2 4 0 0 28 1 2
      4 0 0 25 1 3 4 0 0 28 95 1 3 4 0 0 25
      71 1 2 2 38 0 0 40 2 0 0 6 6 20 2 11
      0 0 0 90 2 1 0 0 6 1 1 20 0 0 51 2 20
      0 0 0 52 2 20 0 6 0 53 2 20 0 0 6 54
      2 23 0 0 0 47 2 23 0 6 0 48 2 23 0 0
      6 49 2 6 0 0 74 1 2 16 0 0 71 1 2 18
      0 0 94 1 2 20 0 71 0 1 2 20 0 74 0 76
      2 23 0 94 0 1 2 18 0 0 0 59 2 18 0 0
```

```
                          6 61 2 18 0 6 0 60)))))
          '|lookupComplete|))
```

2.8.3 The “compilerInfo”

```
(SETQ |$CategoryFrame|
      (|put| '|Equation| '|isFunctor|
             '(((|eval| ($ $ (|List| (|Symbol|)) (|List| |#1|)))
                (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|))
                (ELT $ 31))
               ((|eval| ($ $ (|Symbol|) |#1|))
                (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|))
                (ELT $ 27))
               ((~= ((|Boolean|) $ $)) (|has| |#1| (|SetCategory|))
                (ELT $ NIL))
               ((= ((|Boolean|) $ $)) (|has| |#1| (|SetCategory|))
                (ELT $ 40))
               ((|coerce| ((|OutputForm|) $))
                (|has| |#1| (|SetCategory|)) (ELT $ 44))
               ((|hash| ((|SingleInteger|) $))
                (|has| |#1| (|SetCategory|)) (ELT $ NIL))
               ((|latex| ((|String|) $)) (|has| |#1| (|SetCategory|))
                (ELT $ NIL))
               ((|coerce| ((|Boolean|) $)) (|has| |#1| (|SetCategory|))
                (ELT $ 45))
               ((+ ($ $ $)) (|has| |#1| (|AbelianSemiGroup|))
                (ELT $ 47))
               ((* ($ (|PositiveInteger|) $))
                (|has| |#1| (|AbelianSemiGroup|)) (ELT $ NIL))
               ((|Zero| ($)) (|has| |#1| (|AbelianGroup|))
                (CONST $ 55))
               ((|sample| ($))
                (OR (|has| |#1| (|AbelianGroup|))
                    (|has| |#1| (|Monoid|)))
                (CONST $ NIL))
               ((|zero?| ((|Boolean|) $)) (|has| |#1| (|AbelianGroup|))
                (ELT $ NIL))
               ((* ($ (|NonNegativeInteger|) $))
                (|has| |#1| (|AbelianGroup|)) (ELT $ NIL))
               ((|subtractIfCan| ((|Union| $ "failed") $ $))
                (|has| |#1| (|AbelianGroup|)) (ELT $ NIL))
               ((- ($ $)) (|has| |#1| (|AbelianGroup|)) (ELT $ 51))
               ((- ($ $ $)) (|has| |#1| (|AbelianGroup|)) (ELT $ 52))
               ((* ($ (|Integer|) $)) (|has| |#1| (|AbelianGroup|))
                (ELT $ 76))
               ((* ($ $ $)) (|has| |#1| (|SemiGroup|)) (ELT $ 59))
               ((** ($ $ (|PositiveInteger|)))
                (|has| |#1| (|SemiGroup|)) (ELT $ NIL))
               ((^ ($ $ (|PositiveInteger|)))
                (|has| |#1| (|SemiGroup|)) (ELT $ NIL))
               ((|One| ($)) (|has| |#1| (|Monoid|)) (CONST $ 63))
               ((|one?| ((|Boolean|) $)) (|has| |#1| (|Monoid|))
                (ELT $ NIL))
```

```
((** ($ $ (|NonNegativeInteger|)))
 (|has| |#1| (|Monoid|)) (ELT $ NIL))
((^ ($ $ (|NonNegativeInteger|)))
 (|has| |#1| (|Monoid|)) (ELT $ NIL))
((|recip| ((|Union| $ "failed") $))
 (|has| |#1| (|Monoid|)) (ELT $ 66))
((|inv| ($ $))
 (OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|)))
 (ELT $ 70))
((/ ($ $ $))
 (OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|)))
 (ELT $ 90))
((** ($ $ (|Integer|))) (|has| |#1| (|Group|))
 (ELT $ NIL))
((^ ($ $ (|Integer|))) (|has| |#1| (|Group|))
 (ELT $ NIL))
((|conjugate| ($ $ $)) (|has| |#1| (|Group|))
 (ELT $ NIL))
((|commutator| ($ $ $)) (|has| |#1| (|Group|))
 (ELT $ NIL))
((|characteristic| ((|NonNegativeInteger|)))
 (|has| |#1| (|Ring|)) (ELT $ 73))
((|coerce| ($ (|Integer|))) (|has| |#1| (|Ring|))
 (ELT $ NIL))
((* ($ |#1| $)) (|has| |#1| (|SemiGroup|)) (ELT $ 60))
((* ($ $ |#1|)) (|has| |#1| (|SemiGroup|)) (ELT $ 61))
((|differentiate| ($ $ (|Symbol|)))
 (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
 (ELT $ 85))
((|differentiate| ($ $ (|List| (|Symbol|))))
 (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
 (ELT $ NIL))
((|differentiate|
     ($ $ (|Symbol|) (|NonNegativeInteger|)))
 (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
 (ELT $ NIL))
((|differentiate|
     ($ $ (|List| (|Symbol|))
        (|List| (|NonNegativeInteger|))))
 (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
 (ELT $ NIL))
((D ($ $ (|Symbol|)))
 (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
 (ELT $ NIL))
((D ($ $ (|List| (|Symbol|))))
 (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
 (ELT $ NIL))
((D ($ $ (|Symbol|) (|NonNegativeInteger|)))
 (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
 (ELT $ NIL))
((D ($ $ (|List| (|Symbol|))
       (|List| (|NonNegativeInteger|))))
 (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
 (ELT $ NIL))
```

```
  ((/ ($ $ |#1|)) (|has| |#1| (|Field|)) (ELT $ NIL))
  ((|dimension| ((|CardinalNumber|)))
   (|has| |#1| (|Field|)) (ELT $ 88))
  ((|subst| ($ $ $)) (|has| |#1| (|ExpressionSpace|))
   (ELT $ 93))
  ((|factorAndSplit| ((|List| $) $))
   (|has| |#1| (|IntegralDomain|)) (ELT $ 19))
  ((|rightOne| ((|Union| $ "failed") $))
   (|has| |#1| (|Monoid|)) (ELT $ 68))
  ((|leftOne| ((|Union| $ "failed") $))
   (|has| |#1| (|Monoid|)) (ELT $ 67))
  ((- ($ $ |#1|)) (|has| |#1| (|AbelianGroup|))
   (ELT $ 54))
  ((- ($ |#1| $)) (|has| |#1| (|AbelianGroup|))
   (ELT $ 53))
  ((|rightZero| ($ $)) (|has| |#1| (|AbelianGroup|))
   (ELT $ 8))
  ((|leftZero| ($ $)) (|has| |#1| (|AbelianGroup|))
   (ELT $ 57))
  ((+ ($ $ |#1|)) (|has| |#1| (|AbelianSemiGroup|))
   (ELT $ 49))
  ((+ ($ |#1| $)) (|has| |#1| (|AbelianSemiGroup|))
   (ELT $ 48))
  ((|eval| ($ $ (|List| $)))
   (AND (|has| |#1| (|Evalable| |#1|))
        (|has| |#1| (|SetCategory|)))
   (ELT $ 37))
  ((|eval| ($ $ $))
   (AND (|has| |#1| (|Evalable| |#1|))
        (|has| |#1| (|SetCategory|)))
   (ELT $ 34))
  ((|map| ($ (|Mapping| |#1| |#1|) $)) T (ELT $ 24))
  ((|rhs| (|#1| $)) T (ELT $ 21))
  ((|lhs| (|#1| $)) T (ELT $ 9))
  ((|swap| ($ $)) T (ELT $ 22))
  ((|equation| ($ |#1| |#1|)) T (ELT $ 17))
  ((= ($ |#1| |#1|)) T (ELT $ 20)))
(|addModemap| '|Equation| '(|Equation| |#1|)
    '((|Join| (|Type|)
              (CATEGORY |domain|
                  (SIGNATURE = ($ |#1| |#1|))
                  (SIGNATURE |equation| ($ |#1| |#1|))
                  (SIGNATURE |swap| ($ $))
                  (SIGNATURE |lhs| (|#1| $))
                  (SIGNATURE |rhs| (|#1| $))
                  (SIGNATURE |map|
                      ($ (|Mapping| |#1| |#1|) $))
                  (IF (|has| |#1|
                       (|InnerEvalable| (|Symbol|) |#1|))
                      (ATTRIBUTE
                       (|InnerEvalable| (|Symbol|) |#1|))
                      |noBranch|)
                  (IF (|has| |#1| (|SetCategory|))
                      (PROGN
```

```
          (ATTRIBUTE (|SetCategory|))
          (ATTRIBUTE
           (|CoercibleTo| (|Boolean|)))
          (IF (|has| |#1| (|Evalable| |#1|))
           (PROGN
             (SIGNATURE |eval| ($ $ $))
             (SIGNATURE |eval|
              ($ $ (|List| $))))
           |noBranch|))
        |noBranch|)
    (IF (|has| |#1| (|AbelianSemiGroup|))
        (PROGN
          (ATTRIBUTE (|AbelianSemiGroup|))
          (SIGNATURE + ($ |#1| $))
          (SIGNATURE + ($ $ |#1|)))
        |noBranch|)
    (IF (|has| |#1| (|AbelianGroup|))
        (PROGN
          (ATTRIBUTE (|AbelianGroup|))
          (SIGNATURE |leftZero| ($ $))
          (SIGNATURE |rightZero| ($ $))
          (SIGNATURE - ($ |#1| $))
          (SIGNATURE - ($ $ |#1|)))
        |noBranch|)
    (IF (|has| |#1| (|SemiGroup|))
        (PROGN
          (ATTRIBUTE (|SemiGroup|))
          (SIGNATURE * ($ |#1| $))
          (SIGNATURE * ($ $ |#1|)))
        |noBranch|)
    (IF (|has| |#1| (|Monoid|))
        (PROGN
          (ATTRIBUTE (|Monoid|))
          (SIGNATURE |leftOne|
           ((|Union| $ "failed") $))
          (SIGNATURE |rightOne|
           ((|Union| $ "failed") $)))
        |noBranch|)
    (IF (|has| |#1| (|Group|))
        (PROGN
          (ATTRIBUTE (|Group|))
          (SIGNATURE |leftOne|
           ((|Union| $ "failed") $))
          (SIGNATURE |rightOne|
           ((|Union| $ "failed") $)))
        |noBranch|)
    (IF (|has| |#1| (|Ring|))
        (PROGN
          (ATTRIBUTE (|Ring|))
          (ATTRIBUTE (|BiModule| |#1| |#1|)))
        |noBranch|)
    (IF (|has| |#1| (|CommutativeRing|))
        (ATTRIBUTE (|Module| |#1|))
        |noBranch|)
```

```
                    (IF (|has| |#1| (|IntegralDomain|))
                        (SIGNATURE |factorAndSplit|
                         ((|List| $) $))
                        |noBranch|)
                    (IF (|has| |#1|
                         (|PartialDifferentialRing|
                          (|Symbol|)))
                        (ATTRIBUTE
                         (|PartialDifferentialRing|
                          (|Symbol|)))
                        |noBranch|)
                    (IF (|has| |#1| (|Field|))
                        (PROGN
                          (ATTRIBUTE (|VectorSpace| |#1|))
                          (SIGNATURE / ($ $ $))
                          (SIGNATURE |inv| ($ $)))
                        |noBranch|)
                    (IF (|has| |#1| (|ExpressionSpace|))
                        (SIGNATURE |subst| ($ $ $))
                        |noBranch|)))
  (|Type|))
T '|Equation|
(|put| '|Equation| '|mode|
       '(|Mapping|
             (|Join| (|Type|)
                     (CATEGORY |domain|
                      (SIGNATURE = ($ |#1| |#1|))
                      (SIGNATURE |equation|
                       ($ |#1| |#1|))
                      (SIGNATURE |swap| ($ $))
                      (SIGNATURE |lhs| (|#1| $))
                      (SIGNATURE |rhs| (|#1| $))
                      (SIGNATURE |map|
                       ($ (|Mapping| |#1| |#1|) $))
                      (IF
                       (|has| |#1|
                        (|InnerEvalable| (|Symbol|)
                         |#1|))
                       (ATTRIBUTE
                        (|InnerEvalable| (|Symbol|)
                         |#1|))
                       |noBranch|)
                      (IF (|has| |#1| (|SetCategory|))
                       (PROGN
                         (ATTRIBUTE (|SetCategory|))
                         (ATTRIBUTE
                          (|CoercibleTo| (|Boolean|)))
                         (IF
                          (|has| |#1|
                           (|Evalable| |#1|))
                          (PROGN
                            (SIGNATURE |eval| ($ $ $))
                            (SIGNATURE |eval|
                             ($ $ (|List| $))))
```

```
    |noBranch|))
 |noBranch|)
(IF
 (|has| |#1|
  (|AbelianSemiGroup|))
 (PROGN
   (ATTRIBUTE
    (|AbelianSemiGroup|))
   (SIGNATURE + ($ |#1| $))
   (SIGNATURE + ($ $ |#1|)))
 |noBranch|)
(IF (|has| |#1| (|AbelianGroup|))
 (PROGN
   (ATTRIBUTE (|AbelianGroup|))
   (SIGNATURE |leftZero| ($ $))
   (SIGNATURE |rightZero| ($ $))
   (SIGNATURE - ($ |#1| $))
   (SIGNATURE - ($ $ |#1|)))
 |noBranch|)
(IF (|has| |#1| (|SemiGroup|))
 (PROGN
   (ATTRIBUTE (|SemiGroup|))
   (SIGNATURE * ($ |#1| $))
   (SIGNATURE * ($ $ |#1|)))
 |noBranch|)
(IF (|has| |#1| (|Monoid|))
 (PROGN
   (ATTRIBUTE (|Monoid|))
   (SIGNATURE |leftOne|
    ((|Union| $ "failed") $))
   (SIGNATURE |rightOne|
    ((|Union| $ "failed") $)))
 |noBranch|)
(IF (|has| |#1| (|Group|))
 (PROGN
   (ATTRIBUTE (|Group|))
   (SIGNATURE |leftOne|
    ((|Union| $ "failed") $))
   (SIGNATURE |rightOne|
    ((|Union| $ "failed") $)))
 |noBranch|)
(IF (|has| |#1| (|Ring|))
 (PROGN
   (ATTRIBUTE (|Ring|))
   (ATTRIBUTE
    (|BiModule| |#1| |#1|)))
 |noBranch|)
(IF
 (|has| |#1| (|CommutativeRing|))
 (ATTRIBUTE (|Module| |#1|))
 |noBranch|)
(IF
 (|has| |#1| (|IntegralDomain|))
 (SIGNATURE |factorAndSplit|
```

```
                                         ((|List| $) $))
                                        |noBranch|)
                                       (IF
                                        (|has| |#1|
                                         (|PartialDifferentialRing|
                                          (|Symbol|)))
                                        (ATTRIBUTE
                                         (|PartialDifferentialRing|
                                          (|Symbol|)))
                                        |noBranch|)
                                       (IF (|has| |#1| (|Field|))
                                        (PROGN
                                          (ATTRIBUTE
                                           (|VectorSpace| |#1|))
                                          (SIGNATURE / ($ $ $))
                                          (SIGNATURE |inv| ($ $)))
                                        |noBranch|)
                                       (IF
                                        (|has| |#1| (|ExpressionSpace|))
                                        (SIGNATURE |subst| ($ $ $))
                                        |noBranch|)))
                              (|Type|))
                         |$CategoryFrame|))))
```

2.8.4 The "constructorForm"

```
(|Equation| S)
```

2.8.5 The "constructorKind"

```
|domain|
```

2.8.6 The "constructorModemap"

```
(((|Equation| |#1|)
  (|Join| (|Type|)
          (CATEGORY |domain| (SIGNATURE = ($ |#1| |#1|))
              (SIGNATURE |equation| ($ |#1| |#1|))
              (SIGNATURE |swap| ($ $)) (SIGNATURE |lhs| (|#1| $))
              (SIGNATURE |rhs| (|#1| $))
              (SIGNATURE |map| ($ (|Mapping| |#1| |#1|) $))
              (IF (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|))
                  (ATTRIBUTE (|InnerEvalable| (|Symbol|) |#1|))
                  |noBranch|)
              (IF (|has| |#1| (|SetCategory|))
                  (PROGN
                    (ATTRIBUTE (|SetCategory|))
                    (ATTRIBUTE (|CoercibleTo| (|Boolean|)))
                    (IF (|has| |#1| (|Evalable| |#1|))
                        (PROGN
                          (SIGNATURE |eval| ($ $ $))
```

```
            (SIGNATURE |eval| ($ $ (|List| $))))
          |noBranch|))
      |noBranch|)
    (IF (|has| |#1| (|AbelianSemiGroup|))
      (PROGN
        (ATTRIBUTE (|AbelianSemiGroup|))
        (SIGNATURE + ($ |#1| $))
        (SIGNATURE + ($ $ |#1|)))
      |noBranch|)
    (IF (|has| |#1| (|AbelianGroup|))
      (PROGN
        (ATTRIBUTE (|AbelianGroup|))
        (SIGNATURE |leftZero| ($ $))
        (SIGNATURE |rightZero| ($ $))
        (SIGNATURE - ($ |#1| $))
        (SIGNATURE - ($ $ |#1|)))
      |noBranch|)
    (IF (|has| |#1| (|SemiGroup|))
      (PROGN
        (ATTRIBUTE (|SemiGroup|))
        (SIGNATURE * ($ |#1| $))
        (SIGNATURE * ($ $ |#1|)))
      |noBranch|)
    (IF (|has| |#1| (|Monoid|))
      (PROGN
        (ATTRIBUTE (|Monoid|))
        (SIGNATURE |leftOne| ((|Union| $ "failed") $))
        (SIGNATURE |rightOne| ((|Union| $ "failed") $)))
      |noBranch|)
    (IF (|has| |#1| (|Group|))
      (PROGN
        (ATTRIBUTE (|Group|))
        (SIGNATURE |leftOne| ((|Union| $ "failed") $))
        (SIGNATURE |rightOne| ((|Union| $ "failed") $)))
      |noBranch|)
    (IF (|has| |#1| (|Ring|))
      (PROGN
        (ATTRIBUTE (|Ring|))
        (ATTRIBUTE (|BiModule| |#1| |#1|)))
      |noBranch|)
    (IF (|has| |#1| (|CommutativeRing|))
      (ATTRIBUTE (|Module| |#1|)) |noBranch|)
    (IF (|has| |#1| (|IntegralDomain|))
      (SIGNATURE |factorAndSplit| ((|List| $) $))
      |noBranch|)
    (IF (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
      (ATTRIBUTE (|PartialDifferentialRing| (|Symbol|)))
      |noBranch|)
    (IF (|has| |#1| (|Field|))
      (PROGN
        (ATTRIBUTE (|VectorSpace| |#1|))
        (SIGNATURE / ($ $ $))
        (SIGNATURE |inv| ($ $)))
      |noBranch|)
```

```
             (IF (|has| |#1| (|ExpressionSpace|))
                 (SIGNATURE |subst| ($ $ $)) |noBranch|)))
  (|Type|))
 (T |Equation|))
```

2.8.7 The "constructorCategory"

```
(|Join| (|Type|)
        (CATEGORY |domain| (SIGNATURE = ($ |#1| |#1|))
            (SIGNATURE |equation| ($ |#1| |#1|))
            (SIGNATURE |swap| ($ $)) (SIGNATURE |lhs| (|#1| $))
            (SIGNATURE |rhs| (|#1| $))
            (SIGNATURE |map| ($ (|Mapping| |#1| |#1|) $))
            (IF (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|))
                (ATTRIBUTE (|InnerEvalable| (|Symbol|) |#1|))
                |noBranch|)
            (IF (|has| |#1| (|SetCategory|))
                (PROGN
                  (ATTRIBUTE (|SetCategory|))
                  (ATTRIBUTE (|CoercibleTo| (|Boolean|)))
                  (IF (|has| |#1| (|Evalable| |#1|))
                      (PROGN
                        (SIGNATURE |eval| ($ $ $))
                        (SIGNATURE |eval| ($ $ (|List| $))))
                      |noBranch|))
                |noBranch|)
            (IF (|has| |#1| (|AbelianSemiGroup|))
                (PROGN
                  (ATTRIBUTE (|AbelianSemiGroup|))
                  (SIGNATURE + ($ |#1| $))
                  (SIGNATURE + ($ $ |#1|)))
                |noBranch|)
            (IF (|has| |#1| (|AbelianGroup|))
                (PROGN
                  (ATTRIBUTE (|AbelianGroup|))
                  (SIGNATURE |leftZero| ($ $))
                  (SIGNATURE |rightZero| ($ $))
                  (SIGNATURE - ($ |#1| $))
                  (SIGNATURE - ($ $ |#1|)))
                |noBranch|)
            (IF (|has| |#1| (|SemiGroup|))
                (PROGN
                  (ATTRIBUTE (|SemiGroup|))
                  (SIGNATURE * ($ |#1| $))
                  (SIGNATURE * ($ $ |#1|)))
                |noBranch|)
            (IF (|has| |#1| (|Monoid|))
                (PROGN
                  (ATTRIBUTE (|Monoid|))
                  (SIGNATURE |leftOne| ((|Union| $ "failed") $))
                  (SIGNATURE |rightOne| ((|Union| $ "failed") $)))
                |noBranch|)
            (IF (|has| |#1| (|Group|))
```

```
            (PROGN
              (ATTRIBUTE (|Group|))
              (SIGNATURE |leftOne| ((|Union| $ "failed") $))
              (SIGNATURE |rightOne| ((|Union| $ "failed") $)))
            |noBranch|)
        (IF (|has| |#1| (|Ring|))
            (PROGN
              (ATTRIBUTE (|Ring|))
              (ATTRIBUTE (|BiModule| |#1| |#1|)))
            |noBranch|)
        (IF (|has| |#1| (|CommutativeRing|))
            (ATTRIBUTE (|Module| |#1|)) |noBranch|)
        (IF (|has| |#1| (|IntegralDomain|))
            (SIGNATURE |factorAndSplit| ((|List| $) $)) |noBranch|)
        (IF (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))
            (ATTRIBUTE (|PartialDifferentialRing| (|Symbol|)))
            |noBranch|)
        (IF (|has| |#1| (|Field|))
            (PROGN
              (ATTRIBUTE (|VectorSpace| |#1|))
              (SIGNATURE / ($ $ $))
              (SIGNATURE |inv| ($ $)))
            |noBranch|)
        (IF (|has| |#1| (|ExpressionSpace|))
            (SIGNATURE |subst| ($ $ $)) |noBranch|)))
```

2.8.8 The “sourceFile”

```
"/research/test/int/algebra/EQ.spad"
```

2.8.9 The “modemaps”

```
((= (*1 *1 *2 *2)
    (AND (|isDomain| *1 (|Equation| *2)) (|ofCategory| *2 (|Type|))))
 (|equation| (*1 *1 *2 *2)
     (AND (|isDomain| *1 (|Equation| *2)) (|ofCategory| *2 (|Type|))))
 (|swap| (*1 *1 *1)
         (AND (|isDomain| *1 (|Equation| *2))
              (|ofCategory| *2 (|Type|))))
 (|lhs| (*1 *2 *1)
        (AND (|isDomain| *1 (|Equation| *2))
             (|ofCategory| *2 (|Type|))))
 (|rhs| (*1 *2 *1)
        (AND (|isDomain| *1 (|Equation| *2))
             (|ofCategory| *2 (|Type|))))
 (|map| (*1 *1 *2 *1)
        (AND (|isDomain| *2 (|Mapping| *3 *3))
             (|ofCategory| *3 (|Type|))
             (|isDomain| *1 (|Equation| *3))))
 (|eval| (*1 *1 *1 *1)
         (AND (|ofCategory| *2 (|Evalable| *2))
              (|ofCategory| *2 (|SetCategory|))
```

```
             (|ofCategory| *2 (|Type|))
             (|isDomain| *1 (|Equation| *2))))
(|eval| (*1 *1 *1 *2)
        (AND (|isDomain| *2 (|List| (|Equation| *3)))
             (|ofCategory| *3 (|Evalable| *3))
             (|ofCategory| *3 (|SetCategory|))
             (|ofCategory| *3 (|Type|))
             (|isDomain| *1 (|Equation| *3))))
(+ (*1 *1 *2 *1)
   (AND (|isDomain| *1 (|Equation| *2))
        (|ofCategory| *2 (|AbelianSemiGroup|))
        (|ofCategory| *2 (|Type|))))
(+ (*1 *1 *1 *2)
   (AND (|isDomain| *1 (|Equation| *2))
        (|ofCategory| *2 (|AbelianSemiGroup|))
        (|ofCategory| *2 (|Type|))))
(|leftZero| (*1 *1 *1)
    (AND (|isDomain| *1 (|Equation| *2))
         (|ofCategory| *2 (|AbelianGroup|))
         (|ofCategory| *2 (|Type|))))
(|rightZero| (*1 *1 *1)
    (AND (|isDomain| *1 (|Equation| *2))
         (|ofCategory| *2 (|AbelianGroup|))
         (|ofCategory| *2 (|Type|))))
(- (*1 *1 *2 *1)
   (AND (|isDomain| *1 (|Equation| *2))
        (|ofCategory| *2 (|AbelianGroup|)) (|ofCategory| *2 (|Type|))))
(- (*1 *1 *1 *2)
   (AND (|isDomain| *1 (|Equation| *2))
        (|ofCategory| *2 (|AbelianGroup|)) (|ofCategory| *2 (|Type|))))
(|leftOne| (*1 *1 *1)
    (|partial| AND (|isDomain| *1 (|Equation| *2))
        (|ofCategory| *2 (|Monoid|)) (|ofCategory| *2 (|Type|))))
(|rightOne| (*1 *1 *1)
    (|partial| AND (|isDomain| *1 (|Equation| *2))
        (|ofCategory| *2 (|Monoid|)) (|ofCategory| *2 (|Type|))))
(|factorAndSplit| (*1 *2 *1)
    (AND (|isDomain| *2 (|List| (|Equation| *3)))
         (|isDomain| *1 (|Equation| *3))
         (|ofCategory| *3 (|IntegralDomain|))
         (|ofCategory| *3 (|Type|))))
(|subst| (*1 *1 *1 *1)
         (AND (|isDomain| *1 (|Equation| *2))
              (|ofCategory| *2 (|ExpressionSpace|))
              (|ofCategory| *2 (|Type|))))
(* (*1 *1 *1 *2)
   (AND (|isDomain| *1 (|Equation| *2))
        (|ofCategory| *2 (|SemiGroup|)) (|ofCategory| *2 (|Type|))))
(* (*1 *1 *2 *1)
   (AND (|isDomain| *1 (|Equation| *2))
        (|ofCategory| *2 (|SemiGroup|)) (|ofCategory| *2 (|Type|))))
(/ (*1 *1 *1 *1)
   (OR (AND (|isDomain| *1 (|Equation| *2))
            (|ofCategory| *2 (|Field|)) (|ofCategory| *2 (|Type|)))
```

```
        (AND (|isDomain| *1 (|Equation| *2))
             (|ofCategory| *2 (|Group|)) (|ofCategory| *2 (|Type|)))))
 (|inv| (*1 *1 *1)
        (OR (AND (|isDomain| *1 (|Equation| *2))
                 (|ofCategory| *2 (|Field|))
                 (|ofCategory| *2 (|Type|)))
            (AND (|isDomain| *1 (|Equation| *2))
                 (|ofCategory| *2 (|Group|))
                 (|ofCategory| *2 (|Type|))))))
```

2.8.10 The "operationAlist"

```
((~= (((|Boolean|) $ $) NIL (|has| |#1| (|SetCategory|))))
 (|zero?| (((|Boolean|) $) NIL (|has| |#1| (|AbelianGroup|))))
 (|swap| (($ $) 22))
 (|subtractIfCan|
    (((|Union| $ "failed") $ $) NIL (|has| |#1| (|AbelianGroup|))))
 (|subst| (($ $ $) 93 (|has| |#1| (|ExpressionSpace|))))
 (|sample|
    (($) NIL
     (OR (|has| |#1| (|AbelianGroup|)) (|has| |#1| (|Monoid|))) CONST))
 (|rightZero| (($ $) 8 (|has| |#1| (|AbelianGroup|))))
 (|rightOne| (((|Union| $ "failed") $) 68 (|has| |#1| (|Monoid|))))
 (|rhs| ((|#1| $) 21))
 (|recip| (((|Union| $ "failed") $) 66 (|has| |#1| (|Monoid|))))
 (|one?| (((|Boolean|) $) NIL (|has| |#1| (|Monoid|))))
 (|map| (($ (|Mapping| |#1| |#1|) $) 24)) (|lhs| ((|#1| $) 9))
 (|leftZero| (($ $) 57 (|has| |#1| (|AbelianGroup|))))
 (|leftOne| (((|Union| $ "failed") $) 67 (|has| |#1| (|Monoid|))))
 (|latex| (((|String|) $) NIL (|has| |#1| (|SetCategory|))))
 (|inv| (($ $) 70 (OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|)))))
 (|hash| (((|SingleInteger|) $) NIL (|has| |#1| (|SetCategory|))))
 (|factorAndSplit| (((|List| $) $) 19 (|has| |#1| (|IntegralDomain|))))
 (|eval| (($ $ $) 34
          (AND (|has| |#1| (|Evalable| |#1|))
               (|has| |#1| (|SetCategory|))))
         (($ $ (|List| $)) 37
          (AND (|has| |#1| (|Evalable| |#1|))
               (|has| |#1| (|SetCategory|))))
         (($ $ (|Symbol|) |#1|) 27
          (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|)))
         (($ $ (|List| (|Symbol|)) (|List| |#1|)) 31
          (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|))))
 (|equation| (($ |#1| |#1|) 17))
 (|dimension| (((|CardinalNumber|)) 88 (|has| |#1| (|Field|))))
 (|differentiate|
    (($ $ (|List| (|Symbol|)) (|List| (|NonNegativeInteger|))) NIL
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
    (($ $ (|Symbol|) (|NonNegativeInteger|)) NIL
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
    (($ $ (|List| (|Symbol|))) NIL
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
    (($ $ (|Symbol|)) 85
```

```
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))))
 (|conjugate| (($ $ $) NIL (|has| |#1| (|Group|))))
 (|commutator| (($ $ $) NIL (|has| |#1| (|Group|))))
 (|coerce| (($ (|Integer|)) NIL (|has| |#1| (|Ring|)))
     (((|Boolean|) $) 45 (|has| |#1| (|SetCategory|)))
     (((|OutputForm|) $) 44 (|has| |#1| (|SetCategory|))))
 (|characteristic| (((|NonNegativeInteger|)) 73 (|has| |#1| (|Ring|))))
 (^ (($ $ (|Integer|)) NIL (|has| |#1| (|Group|)))
    (($ $ (|NonNegativeInteger|)) NIL (|has| |#1| (|Monoid|)))
    (($ $ (|PositiveInteger|)) NIL (|has| |#1| (|SemiGroup|))))
 (|Zero| (($) 55 (|has| |#1| (|AbelianGroup|)) CONST))
 (|One| (($) 63 (|has| |#1| (|Monoid|)) CONST))
 (D (($ $ (|List| (|Symbol|)) (|List| (|NonNegativeInteger|))) NIL
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
    (($ $ (|Symbol|) (|NonNegativeInteger|)) NIL
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
    (($ $ (|List| (|Symbol|))) NIL
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
    (($ $ (|Symbol|)) NIL
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))))
 (= (($ |#1| |#1|) 20)
    (((|Boolean|) $ $) 40 (|has| |#1| (|SetCategory|))))
 (/ (($ $ |#1|) NIL (|has| |#1| (|Field|)))
    (($ $ $) 90 (OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|)))))
 (- (($ |#1| $) 53 (|has| |#1| (|AbelianGroup|)))
    (($ $ |#1|) 54 (|has| |#1| (|AbelianGroup|)))
    (($ $ $) 52 (|has| |#1| (|AbelianGroup|)))
    (($ $) 51 (|has| |#1| (|AbelianGroup|))))
 (+ (($ |#1| $) 48 (|has| |#1| (|AbelianSemiGroup|)))
    (($ $ |#1|) 49 (|has| |#1| (|AbelianSemiGroup|)))
    (($ $ $) 47 (|has| |#1| (|AbelianSemiGroup|))))
 (** (($ $ (|Integer|)) NIL (|has| |#1| (|Group|)))
     (($ $ (|NonNegativeInteger|)) NIL (|has| |#1| (|Monoid|)))
     (($ $ (|PositiveInteger|)) NIL (|has| |#1| (|SemiGroup|))))
 (* (($ $ |#1|) 61 (|has| |#1| (|SemiGroup|)))
    (($ |#1| $) 60 (|has| |#1| (|SemiGroup|)))
    (($ $ $) 59 (|has| |#1| (|SemiGroup|)))
    (($ (|Integer|) $) 76 (|has| |#1| (|AbelianGroup|)))
    (($ (|NonNegativeInteger|) $) NIL (|has| |#1| (|AbelianGroup|)))
    (($ (|PositiveInteger|) $) NIL (|has| |#1| (|AbelianSemiGroup|)))))
```

2.8.11 The "superDomain"

2.8.12 The "signaturesAndLocals"

```
((|EQ;subst;3$;43| ($ $ $)) (|EQ;inv;2$;42| ($ $))
 (|EQ;/;3$;41| ($ $ $)) (|EQ;dimension;Cn;40| ((|CardinalNumber|)))
 (|EQ;differentiate;$S$;39| ($ $ (|Symbol|)))
 (|EQ;factorAndSplit;$L;38| ((|List| $) $))
 (|EQ;*;I2$;37| ($ (|Integer|) $))
 (|EQ;characteristic;Nni;36| ((|NonNegativeInteger|)))
 (|EQ;rightOne;$U;35| ((|Union| $ "failed") $))
 (|EQ;leftOne;$U;34| ((|Union| $ "failed") $)) (|EQ;inv;2$;33| ($ $))
```

```
 (|EQ;rightOne;$U;32| ((|Union| $ "failed") $))
 (|EQ;leftOne;$U;31| ((|Union| $ "failed") $))
 (|EQ;recip;$U;30| ((|Union| $ "failed") $)) (|EQ;One;$;29| ($))
 (|EQ;*;$S$;28| ($ $ S)) (|EQ;*;S2$;27| ($ S $))
 (|EQ;*;S2$;26| ($ S $)) (|EQ;*;3$;25| ($ $ $)) (|EQ;-;3$;24| ($ $ $))
 (|EQ;Zero;$;23| ($)) (|EQ;rightZero;2$;22| ($ $))
 (|EQ;leftZero;2$;21| ($ $)) (|EQ;-;$S$;20| ($ $ S))
 (|EQ;-;S2$;19| ($ S $)) (|EQ;-;2$;18| ($ $)) (|EQ;+;$S$;17| ($ $ S))
 (|EQ;+;S2$;16| ($ S $)) (|EQ;+;3$;15| ($ $ $))
 (|EQ;coerce;$B;14| ((|Boolean|) $))
 (|EQ;coerce;$Of;13| ((|OutputForm|) $))
 (|EQ;=;2$B;12| ((|Boolean|) $ $)) (|EQ;eval;$L$;11| ($ $ (|List| $)))
 (|EQ;eval;3$;10| ($ $ $))
 (|EQ;eval;$LL$;9| ($ $ (|List| (|Symbol|)) (|List| S)))
 (|EQ;eval;$SS$;8| ($ $ (|Symbol|) S))
 (|EQ;map;M2$;7| ($ (|Mapping| S S) $)) (|EQ;swap;2$;6| ($ $))
 (|EQ;rhs;$S;5| (S $)) (|EQ;lhs;$S;4| (S $))
 (|EQ;equation;2S$;3| ($ S S)) (|EQ;=;2S$;2| ($ S S))
 (|EQ;factorAndSplit;$L;1| ((|List| $) $)))
```

2.8.13 The "attributes"

```
((|unitsKnown| OR (|has| |#1| (|Ring|)) (|has| |#1| (|Group|)))
 (|rightUnitary| |has| |#1| (|Ring|))
 (|leftUnitary| |has| |#1| (|Ring|)))
```

2.8.14 The "predicates"

```
((|HasCategory| |#1| '(|Field|)) (|HasCategory| |#1| '(|SetCategory|))
 (|HasCategory| |#1| '(|Ring|))
 (|HasCategory| |#1| (LIST '|PartialDifferentialRing| '(|Symbol|)))
 (OR (|HasCategory| |#1| (LIST '|PartialDifferentialRing| '(|Symbol|)))
     (|HasCategory| |#1| '(|Ring|)))
 (|HasCategory| |#1| '(|Group|))
 (|HasCategory| |#1|
     (LIST '|InnerEvalable| '(|Symbol|) (|devaluate| |#1|)))
 (AND (|HasCategory| |#1| (LIST '|Evalable| (|devaluate| |#1|)))
      (|HasCategory| |#1| '(|SetCategory|)))
 (|HasCategory| |#1| '(|IntegralDomain|))
 (|HasCategory| |#1| '(|ExpressionSpace|))
 (OR (|HasCategory| |#1| '(|Field|)) (|HasCategory| |#1| '(|Group|)))
 (OR (|HasCategory| |#1| '(|Group|)) (|HasCategory| |#1| '(|Ring|)))
 (|HasCategory| |#1| '(|CommutativeRing|))
 (OR (|HasCategory| |#1| '(|CommutativeRing|))
     (|HasCategory| |#1| '(|Field|)) (|HasCategory| |#1| '(|Ring|)))
 (OR (|HasCategory| |#1| '(|CommutativeRing|))
     (|HasCategory| |#1| '(|Field|)))
 (|HasCategory| |#1| '(|Monoid|))
 (OR (|HasCategory| |#1| '(|Group|)) (|HasCategory| |#1| '(|Monoid|)))
 (|HasCategory| |#1| '(|SemiGroup|))
 (OR (|HasCategory| |#1| '(|Group|)) (|HasCategory| |#1| '(|Monoid|))
     (|HasCategory| |#1| '(|SemiGroup|)))
```

```
 (|HasCategory| |#1| '(|AbelianGroup|))
 (OR (|HasCategory| |#1| (LIST '|PartialDifferentialRing| '(|Symbol|)))
     (|HasCategory| |#1| '(|AbelianGroup|))
     (|HasCategory| |#1| '(|CommutativeRing|))
     (|HasCategory| |#1| '(|Field|)) (|HasCategory| |#1| '(|Ring|)))
 (OR (|HasCategory| |#1| '(|AbelianGroup|))
     (|HasCategory| |#1| '(|Monoid|)))
 (|HasCategory| |#1| '(|AbelianSemiGroup|))
 (OR (|HasCategory| |#1| (LIST '|PartialDifferentialRing| '(|Symbol|)))
     (|HasCategory| |#1| '(|AbelianGroup|))
     (|HasCategory| |#1| '(|AbelianSemiGroup|))
     (|HasCategory| |#1| '(|CommutativeRing|))
     (|HasCategory| |#1| '(|Field|)) (|HasCategory| |#1| '(|Ring|)))
 (OR (|HasCategory| |#1| (LIST '|PartialDifferentialRing| '(|Symbol|)))
     (|HasCategory| |#1| '(|AbelianGroup|))
     (|HasCategory| |#1| '(|AbelianSemiGroup|))
     (|HasCategory| |#1| '(|CommutativeRing|))
     (|HasCategory| |#1| '(|Field|)) (|HasCategory| |#1| '(|Group|))
     (|HasCategory| |#1| '(|Monoid|)) (|HasCategory| |#1| '(|Ring|))
     (|HasCategory| |#1| '(|SemiGroup|))
     (|HasCategory| |#1| '(|SetCategory|))))
```

2.8.15 The “abbreviation”

```
EQ
```

2.8.16 The “parents”

```
(((|Type|) . T)
 ((|InnerEvalable| (|Symbol|) S) |has| S
  (|InnerEvalable| (|Symbol|) S))
 ((|CoercibleTo| (|Boolean|)) |has| S (|SetCategory|))
 ((|SetCategory|) |has| S (|SetCategory|))
 ((|AbelianSemiGroup|) |has| S (|AbelianSemiGroup|))
 ((|AbelianGroup|) |has| S (|AbelianGroup|))
 ((|SemiGroup|) |has| S (|SemiGroup|)) ((|Monoid|) |has| S (|Monoid|))
 ((|Group|) |has| S (|Group|)) ((|BiModule| S S) |has| S (|Ring|))
 ((|Ring|) |has| S (|Ring|)) ((|Module| S) |has| S (|CommutativeRing|))
 ((|PartialDifferentialRing| (|Symbol|)) |has| S
  (|PartialDifferentialRing| (|Symbol|)))
 ((|VectorSpace| S) |has| S (|Field|)))
```

2.8.17 The “ancestors”

```
(((|AbelianGroup|) |has| S (|AbelianGroup|))
 ((|AbelianMonoid|) |has| S (|AbelianGroup|))
 ((|AbelianSemiGroup|) |has| S (|AbelianSemiGroup|))
 ((|BasicType|) |has| S (|SetCategory|))
 ((|BiModule| S S) |has| S (|Ring|))
 ((|CancellationAbelianMonoid|) |has| S (|AbelianGroup|))
 ((|CoercibleTo| (|OutputForm|)) |has| S (|SetCategory|))
```

```
((|CoercibleTo| (|Boolean|)) |has| S (|SetCategory|))
((|Group|) |has| S (|Group|))
((|InnerEvalable| (|Symbol|) S) |has| S
 (|InnerEvalable| (|Symbol|) S))
((|LeftModule| $) |has| S (|Ring|))
((|LeftModule| S) |has| S (|Ring|))
((|Module| S) |has| S (|CommutativeRing|))
((|Monoid|) |has| S (|Monoid|))
((|PartialDifferentialRing| (|Symbol|)) |has| S
 (|PartialDifferentialRing| (|Symbol|)))
((|RightModule| S) |has| S (|Ring|)) ((|Ring|) |has| S (|Ring|))
((|Rng|) |has| S (|Ring|)) ((|SemiGroup|) |has| S (|SemiGroup|))
((|SetCategory|) |has| S (|SetCategory|)) ((|Type|) . T)
((|VectorSpace| S) |has| S (|Field|)))
```

2.8.18 The "documentation"

```
((|constructor|
     (NIL "Equations as mathematical objects. All properties of the basis
           domain,{} \\spadignore{e.g.} being an abelian group are carried
           over the equation domain,{} by performing the structural operations
            on the left and on the right hand side."))
 (|subst| (($ $ $)
           "\\spad{subst(eq1,{}eq2)} substitutes \\spad{eq2} into both sides
           of \\spad{eq1} the \\spad{lhs} of \\spad{eq2} should be a kernel"))
 (|inv| (($ $)
         "\\spad{inv(x)} returns the multiplicative inverse of \\spad{x}."))
 (/ (($ $ $)
     "\\spad{e1/e2} produces a new equation by dividing the left and right
      hand sides of equations \\spad{e1} and \\spad{e2}."))
 (|factorAndSplit|
     (((|List| $) $)
      "\\spad{factorAndSplit(eq)} make the right hand side 0 and factors the
       new left hand side. Each factor is equated to 0 and put into the
       resulting list without repetitions."))
 (|rightOne|
     (((|Union| $ "failed") $)
      "\\spad{rightOne(eq)} divides by the right hand side.")
     (((|Union| $ "failed") $)
      "\\spad{rightOne(eq)} divides by the right hand side,{} if possible."))
 (|leftOne|
     (((|Union| $ "failed") $)
      "\\spad{leftOne(eq)} divides by the left hand side.")
     (((|Union| $ "failed") $)
      "\\spad{leftOne(eq)} divides by the left hand side,{} if possible."))
 (* (($ $ |#1|)
     "\\spad{eqn*x} produces a new equation by multiplying both sides of
      equation eqn by \\spad{x}.")
    (($ |#1| $)
     "\\spad{x*eqn} produces a new equation by multiplying both sides of
      equation eqn by \\spad{x}."))
 (- (($ $ |#1|)
     "\\spad{eqn-x} produces a new equation by subtracting \\spad{x} from
```

```
     both sides of equation eqn.")
    (($ |#1| $)
     "\\spad{x-eqn} produces a new equation by subtracting both sides of
      equation eqn from \\spad{x}."))
 (|rightZero|
     (($ $) "\\spad{rightZero(eq)} subtracts the right hand side."))
 (|leftZero|
     (($ $) "\\spad{leftZero(eq)} subtracts the left hand side."))
 (+ (($ $ |#1|)
     "\\spad{eqn+x} produces a new equation by adding \\spad{x} to both
     sides of equation eqn.")
    (($ |#1| $)
     "\\spad{x+eqn} produces a new equation by adding \\spad{x} to both
     sides of equation eqn."))
 (|eval| (($ $ (|List| $))
          "\\spad{eval(eqn,{} [x1=v1,{} ... xn=vn])} replaces \\spad{xi}
          by \\spad{vi} in equation \\spad{eqn}.")
         (($ $ $)
          "\\spad{eval(eqn,{} x=f)} replaces \\spad{x} by \\spad{f} in
          equation \\spad{eqn}."))
 (|map| (($ (|Mapping| |#1| |#1|) $)
         "\\spad{map(f,{}eqn)} constructs a new equation by applying
          \\spad{f} to both sides of \\spad{eqn}."))
 (|rhs| ((|#1| $)
         "\\spad{rhs(eqn)} returns the right hand side of equation
          \\spad{eqn}."))
 (|lhs| ((|#1| $)
         "\\spad{lhs(eqn)} returns the left hand side of equation
          \\spad{eqn}."))
 (|swap| (($ $)
          "\\spad{swap(eq)} interchanges left and right hand side of
           equation \\spad{eq}."))
 (|equation|
     (($ |#1| |#1|) "\\spad{equation(a,{}b)} creates an equation."))
 (= (($ |#1| |#1|) "\\spad{a=b} creates an equation.")))
```

2.8.19 The “slotInfo”

```
(|Equation|
    (NIL (~= ((38 0 0) NIL (|has| |#1| (|SetCategory|))))
         (|zero?| ((38 0) NIL (|has| |#1| (|AbelianGroup|))))
         (|swap| ((0 0) 22))
         (|subtractIfCan| ((64 0 0) NIL (|has| |#1| (|AbelianGroup|))))
         (|subst| ((0 0 0) 93 (|has| |#1| (|ExpressionSpace|))))
         (|sample|
             ((0) NIL
              (OR (|has| |#1| (|AbelianGroup|))
                  (|has| |#1| (|Monoid|)))
              CONST))
         (|rightZero| ((0 0) 8 (|has| |#1| (|AbelianGroup|))))
         (|rightOne| ((64 0) 68 (|has| |#1| (|Monoid|))))
         (|rhs| ((6 0) 21))
         (|recip| ((64 0) 66 (|has| |#1| (|Monoid|))))
```

```
(|one?| ((38 0) NIL (|has| |#1| (|Monoid|))))
(|map| ((0 23 0) 24)) (|lhs| ((6 0) 9))
(|leftZero| ((0 0) 57 (|has| |#1| (|AbelianGroup|))))
(|leftOne| ((64 0) 67 (|has| |#1| (|Monoid|))))
(|latex| ((97 0) NIL (|has| |#1| (|SetCategory|))))
(|inv| ((0 0) 70
        (OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|)))))
(|hash| ((96 0) NIL (|has| |#1| (|SetCategory|))))
(|factorAndSplit| ((18 0) 19 (|has| |#1| (|IntegralDomain|))))
(|eval| ((0 0 28 29) 31
         (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|)))
        ((0 0 25 6) 27
         (|has| |#1| (|InnerEvalable| (|Symbol|) |#1|)))
        ((0 0 18) 37
         (AND (|has| |#1| (|Evalable| |#1|))
              (|has| |#1| (|SetCategory|))))
        ((0 0 0) 34
         (AND (|has| |#1| (|Evalable| |#1|))
              (|has| |#1| (|SetCategory|)))))
(|equation| ((0 6 6) 17))
(|dimension| ((86) 88 (|has| |#1| (|Field|))))
(|differentiate|
    ((0 0 25 71) NIL
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
    ((0 0 28 95) NIL
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
    ((0 0 25) 85
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
    ((0 0 28) NIL
     (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))))
(|conjugate| ((0 0 0) NIL (|has| |#1| (|Group|))))
(|commutator| ((0 0 0) NIL (|has| |#1| (|Group|))))
(|coerce| ((38 0) 45 (|has| |#1| (|SetCategory|)))
    ((41 0) 44 (|has| |#1| (|SetCategory|)))
    ((0 74) NIL (|has| |#1| (|Ring|))))
(|characteristic| ((71) 73 (|has| |#1| (|Ring|))))
(^ ((0 0 94) NIL (|has| |#1| (|SemiGroup|)))
   ((0 0 71) NIL (|has| |#1| (|Monoid|)))
   ((0 0 74) NIL (|has| |#1| (|Group|))))
(|Zero| ((0) 55 (|has| |#1| (|AbelianGroup|)) CONST))
(|One| ((0) 63 (|has| |#1| (|Monoid|)) CONST))
(D ((0 0 25 71) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
   ((0 0 28 95) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
   ((0 0 25) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|))))
   ((0 0 28) NIL
    (|has| |#1| (|PartialDifferentialRing| (|Symbol|)))))
(= ((0 6 6) 20) ((38 0 0) 40 (|has| |#1| (|SetCategory|))))
(/ ((0 0 6) NIL (|has| |#1| (|Field|)))
   ((0 0 0) 90
    (OR (|has| |#1| (|Field|)) (|has| |#1| (|Group|)))))
(- ((0 0 6) 54 (|has| |#1| (|AbelianGroup|)))
```

```
            ((0 6 0) 53 (|has| |#1| (|AbelianGroup|)))
            ((0 0 0) 52 (|has| |#1| (|AbelianGroup|)))
            ((0 0) 51 (|has| |#1| (|AbelianGroup|))))
         (+ ((0 0 6) 49 (|has| |#1| (|AbelianSemiGroup|)))
            ((0 6 0) 48 (|has| |#1| (|AbelianSemiGroup|)))
            ((0 0 0) 47 (|has| |#1| (|AbelianSemiGroup|))))
         (** ((0 0 94) NIL (|has| |#1| (|SemiGroup|)))
             ((0 0 71) NIL (|has| |#1| (|Monoid|)))
             ((0 0 74) NIL (|has| |#1| (|Group|))))
         (* ((0 6 0) 60 (|has| |#1| (|SemiGroup|)))
            ((0 0 6) 61 (|has| |#1| (|SemiGroup|)))
            ((0 0 0) 59 (|has| |#1| (|SemiGroup|)))
            ((0 94 0) NIL (|has| |#1| (|AbelianSemiGroup|)))
            ((0 74 0) 76 (|has| |#1| (|AbelianGroup|)))
            ((0 71 0) NIL (|has| |#1| (|AbelianGroup|))))))
```

2.8.20 The "index"

```
(("slot1Info" 0 32444) ("documentation" 0 29640) ("ancestors" 0 28691)
 ("parents" 0 28077) ("abbreviation" 0 28074) ("predicates" 0 25442)
 ("attributes" 0 25304) ("signaturesAndLocals" 0 23933)
 ("superDomain" 0 NIL) ("operationAlist" 0 20053) ("modemaps" 0 17216)
 ("sourceFile" 0 17179) ("constructorCategory" 0 15220)
 ("constructorModemap" 0 13215) ("constructorKind" 0 13206)
 ("constructorForm" 0 13191) ("compilerInfo" 0 4433)
 ("loadTimeStuff" 0 20))
```

Chapter 3

Compiler top level

3.1 Spad Program Representation

From Davenport[Dave84a] and Dos Reis et al.[Dosr11]

The Spad programming language is strongly typed. Yet, it allows for runtime instantiation of domains and categories. Consequently, categories and domains are both compile-time and runtime objects. From now on, we will discuss only the representation of category objects. Domains and packages are similarly represented, with some variations to attend to data specific to domains.

0	CategoryForm	
1	ExportInfoList	
2	AttributeList	
3	(Category)	
4	0	PrincipalAncestorList
	1	ExtendedCategoryList
	2	DomainInfoList
5	UsedDomainList	
$\vdots$	vdots	

Figure: Layout of category objects

A category object is represented as a large heterogeneous tuple as shown in the figure above. Its components have the following meaning:

- slot 0 holds the canonical category form of the expression whose evaluation produces the category object under consideration.
- slot 1 holds a list of function signatures exported by the category
- slot 2 holds a list of attributes and the condition under which they hold
- slot 3 always contains the form '(category)'. It serves as a runtime type checking tag
- slot 4 contains 3 parts:
  - a list of principal ancestor category forms
  - a list of directly extended category forms

  - a list of domains explicitly used in that category

- slot 5 holds the list of all domain forms mentioned in the exported signatures
- each of the slots 6 and onwards holds either runtime information about a specific exported signature, or a pointer to a domain object or a category object.

3.2 Global Data Structures

3.3 Pratt Parsing

Parsing involves understanding the association of symbols and operators. Vaughn Pratt [Prat73] poses the question "Given a substring AEB where A takes a right argument, B a left, and E is an expression, does E associate with A or B?".

Floyd [Floy63] associates a precedence with operators, storing them in a table, called "binding powers". The expression E would associate with the argument position having the highest binding power. This leads to a large set of numbers, one for every situation.

Pratt assigns data types to "classes" and then creates a total order on the classes. He lists, in ascending order, Outcomes, Booleans, Graphs (trees, lists, etc), Strings, Algebraics (e.g. Integer, complex numbers, polynomials, real arrays) and references (e.g. the left hand side of assignments). Thus, Strings ¡ References. The key restriction is "that the class of the type at any argument that might participate in an association problem not be less than the class of the data type of the result of the function taking that argument".

For a less-than comparision ("<") the argument types are Algebraics but the result type is Boolean. Since Algebraics are greater than Boolean we can associate the Algebraics together and apply them as arguments to the Boolean.

In more detail, there an "association" is a function of 4 types:

- a_A – The data type of the right argument
- r_A – The return type of the right argument
- a_B – The data type of the left argument
- r_B – The return type of the left argument

Note that the return types might depend on the type of the expression E. If all 4 are of the same class then the association is to the left.

Using these ideas and given the restriction above, Pratt proves that every association problem has at most one solution consistant with the data types of the associated operators.

Pratt proves that there exists an assignment of integers to the argument positions of each token in the language such that the correct association, if any, is always in the direction of the argument position with the larger number, with ties being broken to the left.

To construct the proper numbers, first assign even integers to the data type classes. Then to each argument position assign an integer lying strictly (where possible) between the integers corresponding to the classes of the argument and result types.

For tokens like "and", "or", +, $*$, and the Booleans and Algebraics can be subdivided into pseudo-classes so that

terms $<$ factors $<$ primaries

Then + is defined over terms, ∗ over factors, and over primaries with coercions allowed from primaries to factors to terms. To be consistent with Algol, the primaries should be a right associative class (e.g. xyz)

3.4)compile

This is the implementation of the)compile command.

You use this command to invoke the new Axiom library compiler or the old Axiom system compiler. The `)compile` system command is actually a combination of Axiom processing and a call to the Aldor compiler. It is performing double-duty, acting as a front-end to both the Aldor compiler and the old Axiom system compiler. (The old Axiom system compiler was written in Lisp and was an integral part of the Axiom environment. The Aldor compiler is written in C and executed by the operating system when called from within Axiom.)

User Level Required: compiler

Command Syntax:

`)compile`

`)compile` *fileName*

`)compile` *fileName*`.spad`

`)compile` *directory/fileName*`.spad`

`)compile` *fileName* `)old`

`)compile` *fileName* `)translate`

`)compile` *fileName* `)quiet`

`)compile` *fileName* `)noquiet`

`)compile` *fileName* `)moreargs`

`)compile` *fileName* `)onlyargs`

`)compile` *fileName* `)break`

`)compile` *fileName* `)nobreak`

`)compile` *fileName* `)library`

`)compile` *fileName* `)nolibrary`

`)compile` *fileName* `)vartrace`

`)compile` *fileName* `)constructor` *nameOrAbbrev*

Command Description:

The first thing `)compile` does is look for a source code filename among its arguments. Thus

```
)compile mycode.spad
)compile /u/jones/mycode.spad
)compile mycode
```

all invoke `)compiler` on the file `/u/jones/mycode.spad` if the current Axiom working directory is `/u/jones.` (Recall that you can set the working directory via the `)cd` command. If you don't set it explicitly, it is the directory from which you started Axiom.)

If you omit the file extension, the command looks to see if you have specified the `)new` or

`)old` option. If you have given one of these options, the corresponding compiler is used.

The command first looks in the standard system directories for files with extension *.as, .ao* and *.al* and then files with extension *.spad*. The first file found has the appropriate compiler invoked on it. If the command cannot find a matching file, an error message is displayed and the command terminates.

The first thing `)compile` does is look for a source code filename among its arguments. Thus

```
)compile mycode
)co mycode
)co mycode.spad
```

all invoke `)compiler` on the file `/u/jones/mycode.spad` if the current Axiom working directory is `/u/jones.` Recall that you can set the working directory via the `)cd` command. If you don't set it explicitly, it is the directory from which you started Axiom.

This is frequently all you need to compile your file.

This simple command:

1. Invokes the Spad compiler and produces Lisp output.
2. Calls the Lisp compiler if the compilation was successful.
3. Uses the `)library` command to tell Axiom about the contents of your compiled file and arrange to have those contents loaded on demand.

Should you not want the `)library` command automatically invoked, call `)compile` with the `)nolibrary` option. For example,

```
)compile mycode )nolibrary
```

By default, the `)library` system command *exposes* all domains and categories it processes. This means that the Axiom intepreter will consider those domains and categories when it is trying to resolve a reference to a function. Sometimes domains and categories should not be exposed. For example, a domain may just be used privately by another domain and may not be meant for top-level use. The `)library` command should still be used, though, so that the code will be loaded on demand. In this case, you should use the `)nolibrary` option on `)compile` and the `)noexpose` option in the `)library` command. For example,

```
)compile mycode )nolibrary
)library mycode )noexpose
```

Once you have established your own collection of compiled code, you may find it handy to use the `)dir` option on the `)library` command. This causes `)library` to process all compiled code in the specified directory. For example,

```
)library )dir /u/jones/quantum
```

You must give an explicit directory after `)dir`, even if you want all compiled code in the current working directory processed, e.g.

```
)library )dir .
```

3.4.1 Spad compiler

This command compiles files with file extension `.spad` with the Spad system compiler.

The `)translate` option is used to invoke a special version of the old system compiler that will translate a *.spad* file to a *.as* file. That is, the *.spad* file will be parsed and analyzed and a file using the new syntax will be created.

By default, the *.as* file is created in the same directory as the *.spad* file. If that directory is not writable, the current directory is used. If the current directory is not writable, an error message is given and the command terminates. Note that `)translate` implies the `)old` option so the file extension can safely be omitted. If `)translate` is given, all other options are ignored. Please be aware that the translation is not necessarily one hundred percent complete or correct. You should attempt to compile the output with the Aldor compiler and make any necessary corrections.

You can compile category, domain, and package constructors contained in files with file extension *.spad*. You can compile individual constructors or every constructor in a file.

The full filename is remembered between invocations of this command and `)edit` commands. The sequence of commands

```
)compile matrix.spad
)edit
)compile
```

will call the compiler, edit, and then call the compiler again on the file **matrix.spad.** If you do not specify a *directory,* the working current directory is searched for the file. If the file is not found, the standard system directories are searched.

If you do not give any options, all constructors within a file are compiled. Each constructor should have an `)abbreviation` command in the file in which it is defined. We suggest that you place the `)abbreviation` commands at the top of the file in the order in which the constructors are defined.

The `)library` option causes directories containing the compiled code for each constructor to be created in the working current directory. The name of such a directory consists of the constructor abbreviation and the **.nrlib** file extension. For example, the directory containing the compiled code for the `MATRIX` constructor is called **MATRIX.nrlib.** The `)nolibrary` option says that such files should not be created. The default is `)library.`

The `)vartrace` option causes the compiler to generate extra code for the constructor to support conditional tracing of variable assignments. Without this option, this code is suppressed and one cannot use the `)vars` option for the trace command.

The `)constructor` option is used to specify a particular constructor to compile. All other constructors in the file are ignored. The constructor name or abbreviation follows `)constructor.` Thus either

```
)compile matrix.spad )constructor RectangularMatrix
```

or

```
)compile matrix.spad )constructor RMATRIX
```

compiles the `RectangularMatrix` constructor defined in **matrix.spad.**

The `)break` and `)nobreak` options determine what the spad compiler does when it encounters an error. `)break` is the default and it indicates that processing should stop at the first error. The value of the `)set break` variable then controls what happens.

3.5 Operator Precedence Table Initialization

```
; PURPOSE: This file sets up properties which are used by the Boot lexical
```

```
;          analyzer for bottom-up recognition of operators.  Also certain
;          other character-class definitions are included, as well as
;          table accessing functions.
;
; ORGANIZATION: Each section is organized in terms of Creation and Access code.
;
;               1. Led and Nud Tables
;               2. GLIPH  Table
;               3. RENAMETOK Table
;               4. GENERIC Table
;               5. Character syntax class predicates
```

3.5.1 LED and NUD Tables

```
; **** 1. LED and NUD Tables

; ** TABLE PURPOSE

; Led and Nud have to do with operators. An operator with a Led property takes
; an operand on its left (infix/suffix operator).

; An operator with a Nud takes no operand on its left (prefix/nilfix).
; Some have both (e.g. - ).  This terminology is from the Pratt parser.
; The translator for Scratchpad II is a modification of the Pratt parser which
; branches to special handlers when it is most convenient and practical to
; do so (Pratt's scheme cannot handle local contexts very easily).

; Both LEDs and NUDs have right and left binding powers.  This is meaningful
; for prefix and infix operators.  These powers are stored as the values of
; the LED and NUD properties of an atom, if the atom has such a property.
; The format is:

;       <Operator Left-Binding-Power  Right-Binding-Power <Special-Handler>>

; where the Special-Handler is the name of a function to be evaluated when that
; keyword is encountered.

; The default values of Left and Right Binding-Power are NIL.  NIL is a
; legitimate value signifying no precedence.  If the Special-Handler is NIL,
; this is just an ordinary operator (as opposed to a surfix operator like
; if-then-else).
;
; The Nud value gives the precedence when the operator is a prefix op.
; The Led value gives the precedence when the operator is an infix op.
; Each op has 2 priorities, left and right.
; If the right priority of the first is greater than or equal to the
; left priority of the second then collect the second operator into
; the right argument of the first operator.
```

— LEDNUDTables —

```
; ** TABLE CREATION
```

```
(defun makenewop (x y) (makeop x y '|PARSE-NewKEY|))

(defun makeop (x y keyname)
  (if (or (not (cdr x)) (numberp (second x)))
      (setq x (cons (first x) x)))
  (if (and (alpha-char-p (elt (princ-to-string (first x)) 0))
           (not (member (first x) (eval keyname))))
      (set keyname (cons (first x) (eval keyname))))
  (put (first x) y x)
  (second x))

(setq |PARSE-NewKEY| nil) ;;list of keywords

(mapcar #'(LAMBDA(J) (MAKENEWOP J '|Led|))
        '((* 800 801)   (|rem| 800 801)   (|mod| 800 801)
          (|quo| 800 801)   (|div| 800 801)
          (/ 800 801)    (** 900 901)  (^ 900 901)
          (|exquo| 800 801) (+ 700 701)
          (\- 700 701)    (\-\> 1001 1002)  (\<\- 1001 1002)
          (\: 996 997)    (\:\: 996 997)
          (\@ 996 997)    (|pretend| 995 996)
          (\.)            (\! \! 1002 1001)
          (\, 110 111)
          (\; 81 82 (|PARSE-SemiColon|))
          (\< 400 400)    (\> 400 400)
          (\<\< 400 400)  (\>\> 400 400)
          (\<= 400 400)   (\>= 400 400)
          (= 400 400)     (^= 400 400)
          (\~= 400 400)
          (|in| 400 400)    (|case| 400 400)
          (|add| 400 120)   (|with| 2000 400 (|PARSE-InfixWith|))
          (|has| 400 400)
          (|where| 121 104)     ; must be 121 for SPAD, 126 for boot--> nboot
          (|when| 112 190)
          (|otherwise| 119 190 (|PARSE-Suffix|))
          (|is| 400 400)    (|isnt| 400 400)
          (|and| 250 251)   (|or| 200 201)
          (/\\ 250 251)   (\\/ 200 201)
          (\.\. SEGMENT 401 699 (|PARSE-Seg|))
          (=\> 123 103)
          (+-\> 995 112)
          (== DEF 122 121)
          (==\> MDEF 122 121)
          (\| 108 111)                          ;was 190 190
          (\:- LETD 125 124) (\:= LET 125 124)))

(mapcar #'(LAMBDA (J) (MAKENEWOP J `|Nud|))
        '((|for| 130 350 (|PARSE-Loop|))
          (|while| 130 190 (|PARSE-Loop|))
          (|until| 130 190 (|PARSE-Loop|))
          (|repeat| 130 190 (|PARSE-Loop|))
          (|import| 120 0 (|PARSE-Import|) )
          (|unless|)
```

```
          (|add| 900 120)
          (|with| 1000 300 (|PARSE-With|))
          (|has| 400 400)
          (\- 701 700)  ; right-prec. wants to be -1 + left-prec
;;        (\+ 701 700)
          (\# 999 998)
          (\! 1002 1001)
          (\' 999 999 (|PARSE-Data|))
          (\<\< 122 120 (|PARSE-LabelExpr|))
          (\>\>)
          (^ 260 259 NIL)
          (\-\> 1001 1002)
          (\: 194 195)
          (|not| 260 259 NIL)
          (\~ 260 259 nil)
          (\= 400 700)
          (|return| 202 201 (|PARSE-Return|))
          (|leave| 202 201 (|PARSE-Leave|))
          (|exit| 202 201 (|PARSE-Exit|))
          (|from|)
          (|iterate|)
          (|yield|)
          (|if| 130 0 (|PARSE-Conditional|))    ; was 130
          (\| 0 190)
          (|suchthat|)
          (|then| 0 114)
          (|else| 0 114)))
```

————-

3.6 Gliph Table

Gliphs are symbol clumps. The gliph property of a symbol gives the tree describing the tokens which begin with that symbol. The token reader uses the gliph property to determine the longest token. Thus := is read as one token not as : followed by =.

— **GLIPHTable** —

```
(mapcar #'(lambda (x) (put (car x) 'gliph (cdr x)))
        `(
          ( \| (\))        )
          ( *  (*)         )
          ( \( (<) (\|)    )
          ( +  (- (>))     )
          ( -  (>)         )
          ( <  (=) (<)     )
     ;;     ( /  (\\)        ) breaks */xxx
          ( \\ (/)         )
          ( >  (=) (>) (\)))
          ( =  (= (>)) (>) )
          ( \. (\.)        )
          ( ^  (=)         )
```

```
( \~ (=)         )
( \: (=) (-) (\:))))
```

———-

3.6.1 Rename Token Table

RENAMETOK defines alternate token strings which can be used for different keyboards which define equivalent tokens.

— **RENAMETOKTable** —

```
(mapcar
  #'(lambda (x) (put (car x) 'renametok (cadr x)) (makenewop x nil))
        '((\(\| \[)                     ; (| |) means []
          (\|\) \])
          (\(< \{)                      ; (< >) means {}
          (>\) \})))
```

———-

3.6.2 Generic function table

GENERIC operators be suffixed by $ qualifications in SPAD code. $ is then followed by a domain label, such as I for Integer, which signifies which domain the operator refers to. For example `+$Integer` is + for Integers.

— **GENERICTable** —

```
(mapcar #'(lambda (x) (put x 'generic 'true))
        '(- = * |rem| |mod| |quo| |div| / ** |exquo| + - < > <= >= ^= ))
```

———-

3.7 Giant steps, Baby steps

We will walk through the compiler with the EQ.spad example using a Giant-steps, Baby-steps approach. That is, we will show the large scale (Giant) transformations at each stage of compilation and discuss the details (Baby) in subsequent chapters.

Chapter 4

The Parser

4.1 EQ.spad

We will explain the compilation function using the file `EQ.spad`. We trace the execution of the various functions to understand the actual call parameters and results returned. The `EQ.spad` file is:

```
)abbrev domain EQ Equation
--FOR THE BENEFIT  OF LIBAXO GENERATION
++ Author: Stephen M. Watt, enhancements by Johannes Grabmeier
++ Date Created: April 1985
++ Date Last Updated: June 3, 1991; September 2, 1992
++ Basic Operations: =
++ Related Domains:
++ Also See:
++ AMS Classifications:
++ Keywords: equation
++ Examples:
++ References:
++ Description:
++ Equations as mathematical objects.  All properties of the basis domain,
++ e.g. being an abelian group are carried over the equation domain, by
++ performing the structural operations on the left and on the
++ right hand side.
--   The interpreter translates "=" to "equation".  Otherwise, it will
--   find a modemap for "=" in the domain of the arguments.

Equation(S: Type): public == private where
  Ex ==> OutputForm
  public ==> Type with
    "=": (S, S) -> $
        ++ a=b creates an equation.
    equation: (S, S) -> $
        ++ equation(a,b) creates an equation.
    swap: $ -> $
        ++ swap(eq) interchanges left and right hand side of equation eq.
    lhs: $ -> S
```

```
    ++ lhs(eqn) returns the left hand side of equation eqn.
rhs: $ -> S
    ++ rhs(eqn) returns the right hand side of equation eqn.
map: (S -> S, $) -> $
    ++ map(f,eqn) constructs a new equation by applying f to both
    ++ sides of eqn.
if S has InnerEvalable(Symbol,S) then
         InnerEvalable(Symbol,S)
if S has SetCategory then
    SetCategory
    CoercibleTo Boolean
    if S has Evalable(S) then
       eval: ($, $) -> $
           ++ eval(eqn, x=f) replaces x by f in equation eqn.
       eval: ($, List $) -> $
           ++ eval(eqn, [x1=v1, ... xn=vn]) replaces xi by vi in equation eqn.
if S has AbelianSemiGroup then
    AbelianSemiGroup
    "+": (S, $) -> $
        ++ x+eqn produces a new equation by adding x to both sides of
        ++ equation eqn.
    "+": ($, S) -> $
        ++ eqn+x produces a new equation by adding x to  both sides of
        ++ equation eqn.
if S has AbelianGroup then
    AbelianGroup
    leftZero : $ -> $
      ++ leftZero(eq) subtracts the left hand side.
    rightZero : $ -> $
      ++ rightZero(eq) subtracts the right hand side.
    "-": (S, $) -> $
        ++ x-eqn produces a new equation by subtracting both sides of
        ++ equation eqn from x.
    "-": ($, S) -> $
        ++ eqn-x produces a new equation by subtracting x from  both sides of
        ++ equation eqn.
if S has SemiGroup then
    SemiGroup
    "*": (S, $) -> $
        ++ x*eqn produces a new equation by multiplying both sides of
        ++ equation eqn by x.
    "*": ($, S) -> $
        ++ eqn*x produces a new equation by multiplying both sides of
        ++ equation eqn by x.
if S has Monoid then
    Monoid
    leftOne : $ -> Union($,"failed")
      ++ leftOne(eq) divides by the left hand side, if possible.
    rightOne : $ -> Union($,"failed")
      ++ rightOne(eq) divides by the right hand side, if possible.
if S has Group then
    Group
    leftOne : $ -> Union($,"failed")
      ++ leftOne(eq) divides by the left hand side.
```

```
        rightOne : $ -> Union($,"failed")
          ++ rightOne(eq) divides by the right hand side.
    if S has Ring then
      Ring
      BiModule(S,S)
    if S has CommutativeRing then
      Module(S)
      --Algebra(S)
    if S has IntegralDomain then
      factorAndSplit : $ -> List $
        ++ factorAndSplit(eq) make the right hand side 0 and
        ++ factors the new left hand side. Each factor is equated
        ++ to 0 and put into the resulting list without repetitions.
    if S has PartialDifferentialRing(Symbol) then
      PartialDifferentialRing(Symbol)
    if S has Field then
      VectorSpace(S)
      "/": ($, $) -> $
          ++ e1/e2 produces a new equation by dividing the left and right
          ++ hand sides of equations e1 and e2.
      inv: $ -> $
          ++ inv(x) returns the multiplicative inverse of x.
    if S has ExpressionSpace then
        subst: ($, $) -> $
             ++ subst(eq1,eq2) substitutes eq2 into both sides of eq1
             ++ the lhs of eq2 should be a kernel

  private ==> add
    Rep := Record(lhs: S, rhs: S)
    eq1,eq2: $
    s : S
    if S has IntegralDomain then
        factorAndSplit eq ==
          (S has factor : S -> Factored S) =>
            eq0 := rightZero eq
            [equation(rcf.factor,0) for rcf in factors factor lhs eq0]
          [eq]
    l:S = r:S      == [l, r]
    equation(l, r) == [l, r]    -- hack!  See comment above.
    lhs eqn        == eqn.lhs
    rhs eqn        == eqn.rhs
    swap eqn     == [rhs eqn, lhs eqn]
    map(fn, eqn)   == equation(fn(eqn.lhs), fn(eqn.rhs))

    if S has InnerEvalable(Symbol,S) then
        s:Symbol
        ls:List Symbol
        x:S
        lx:List S
        eval(eqn,s,x) == eval(eqn.lhs,s,x) = eval(eqn.rhs,s,x)
        eval(eqn,ls,lx) == eval(eqn.lhs,ls,lx) = eval(eqn.rhs,ls,lx)
    if S has Evalable(S) then
        eval(eqn1:$, eqn2:$):$ ==
           eval(eqn1.lhs, eqn2 pretend Equation S) =
```

```
            eval(eqn1.rhs, eqn2 pretend Equation S)
      eval(eqn1:$, leqn2:List $):$ ==
         eval(eqn1.lhs, leqn2 pretend List Equation S) =
            eval(eqn1.rhs, leqn2 pretend List Equation S)
  if S has SetCategory then
      eq1 = eq2 == (eq1.lhs = eq2.lhs)@Boolean and
                   (eq1.rhs = eq2.rhs)@Boolean
      coerce(eqn:$):Ex == eqn.lhs::Ex = eqn.rhs::Ex
      coerce(eqn:$):Boolean == eqn.lhs = eqn.rhs
  if S has AbelianSemiGroup then
      eq1 + eq2 == eq1.lhs + eq2.lhs = eq1.rhs + eq2.rhs
      s + eq2 == [s,s] + eq2
      eq1 + s == eq1 + [s,s]
  if S has AbelianGroup then
      - eq == (- lhs eq) = (-rhs eq)
      s - eq2 == [s,s] - eq2
      eq1 - s == eq1 - [s,s]
      leftZero eq == 0 = rhs eq - lhs eq
      rightZero eq == lhs eq - rhs eq = 0
      0 == equation(0$S,0$S)
      eq1 - eq2 == eq1.lhs - eq2.lhs = eq1.rhs - eq2.rhs
  if S has SemiGroup then
      eq1:$ * eq2:$ == eq1.lhs * eq2.lhs = eq1.rhs * eq2.rhs
      l:S   * eqn:$ == l       * eqn.lhs = l       * eqn.rhs
      l:S * eqn:$  ==  l * eqn.lhs    =    l * eqn.rhs
      eqn:$ * l:S  ==  eqn.lhs * l    =    eqn.rhs * l
      -- We have to be a bit careful here: raising to a +ve integer is OK
      -- (since it's the equivalent of repeated multiplication)
      -- but other powers may cause contradictions
      -- Watch what else you add here! JHD 2/Aug 1990
  if S has Monoid then
      1 == equation(1$S,1$S)
      recip eq ==
        (lh := recip lhs eq) case "failed" => "failed"
        (rh := recip rhs eq) case "failed" => "failed"
        [lh :: S, rh :: S]
      leftOne eq ==
        (re := recip lhs eq) case "failed" => "failed"
        1 = rhs eq * re
      rightOne eq ==
        (re := recip rhs eq) case "failed" => "failed"
        lhs eq * re = 1
  if S has Group then
      inv eq == [inv lhs eq, inv rhs eq]
      leftOne eq == 1 = rhs eq * inv rhs eq
      rightOne eq == lhs eq * inv rhs eq = 1
  if S has Ring then
      characteristic() == characteristic()$S
      i:Integer * eq:$ == (i::S) * eq
  if S has IntegralDomain then
      factorAndSplit eq ==
        (S has factor : S -> Factored S) =>
          eq0 := rightZero eq
          [equation(rcf.factor,0) for rcf in factors factor lhs eq0]
```

```
          (S has Polynomial Integer) =>
            eq0 := rightZero eq
            MF ==> MultivariateFactorize(Symbol, IndexedExponents Symbol, _
               Integer, Polynomial Integer)
            p : Polynomial Integer := (lhs eq0) pretend Polynomial Integer
            [equation((rcf.factor) pretend S,0) for rcf in factors factor(p)$MF]
          [eq]
    if S has PartialDifferentialRing(Symbol) then
        differentiate(eq:$, sym:Symbol):$ ==
           [differentiate(lhs eq, sym), differentiate(rhs eq, sym)]
    if S has Field then
        dimension() == 2 :: CardinalNumber
        eq1:$ / eq2:$ == eq1.lhs / eq2.lhs = eq1.rhs / eq2.rhs
        inv eq == [inv lhs eq, inv rhs eq]
    if S has ExpressionSpace then
        subst(eq1,eq2) ==
            eq3 := eq2 pretend Equation S
            [subst(lhs eq1,eq3),subst(rhs eq1,eq3)]
```

4.2 boot transformations

4.2.1 defun string2BootTree

[new2OldLisp p78]
[def-rename p527]
[boot-line-stack p??]
[xtokenreader p??]
[line-handler p??]
[$boot p??]
[$spad p515]

— defun string2BootTree —

```
(defun string2BootTree (s)
 (init-boot/spad-reader)
 (let* ((boot-line-stack (list (cons 1 s)))
        ($boot t)
        ($spad nil)
        (xtokenreader 'get-boot-token)
        (line-handler 'next-boot-line)
        (parseout (progn (|PARSE-Expression|) (pop-stack-1))))
 (declare (special boot-line-stack $boot $spad xtokenreader line-handler))
 (def-rename (new2OldLisp parseout))))
```

———-

4.2.2 defun new2OldLisp

[new2OldTran p78]
[postTransform p337]

— defun new2OldLisp —

```
(defun new2OldLisp (x)
 (new2OldTran (postTransform x)))
```

———-

4.2.3 defun new2OldTran

[dcq p??]
[new2OldTran p78]
[newDef2Def p79]
[newIf2Cond p79]
[newConstruct p80]
[$new2OldRenameAssoc p??]

— defun new2OldTran —

```
(defun new2OldTran (x)
 (prog (tmp1 tmp2 tmp3 tmp4 tmp5 tmp6 a b c d)
 (declare (special |$new2OldRenameAssoc|))
  (return
   (prog nil
    (if (atom x)
     (return (let ((y (assoc x |$new2OldRenameAssoc|)))
                    (if y (cdr y) x))))
    (if (and (dcq (tmp1 a b . tmp2) x)
             (null tmp2)
             (eq tmp1 '|where|)
             (dcq (tmp3 . tmp4) b)
             (dcq ((tmp5 d . tmp6) . c) (reverse tmp4))
             (null tmp6)
             (eq tmp5 '|exit|)
             (eq tmp3 'seq)
             (or (setq c (nreverse c)) t))
        (return
         `(|where| ,(new2OldTran a) ,@(new2OldTran c)
                    ,(new2OldTran d))))
     (return
      (case (car x)
       (quote x)
       (def (newDef2Def x))
       (if (newIf2Cond x))
       ; construct === #'list (see patches.lisp) TPD 12/2011
       (|construct| (newConstruct (new2OldTran (cdr x))))
       (t `(,(new2OldTran (car x)) . ,(new2OldTran (cdr x)))))))))
```

———-

4.2.4 defun newIf2Cond

[letError p??]
[new2OldTran p78]

— defun newIf2Cond —

```
(defun newIf2Cond (cond-expr)
  (if (not (and (= (length cond-expr) 4) (eq (car cond-expr) 'if)))
    (letError "(IF,a,b,c)" cond-expr)
    (let ((a (second cond-expr))
          (b (third cond-expr))
          (c (fourth cond-expr)))
      (setq a (new2OldTran a) b (new2OldTran b) c (new2OldTran c))
      (if (eq c '|noBranch|)
        `(if ,a ,b))
        `(if ,a ,b ,c))))
```

———-

(defun letError (form val) (—systemError— (format nil ” S is not matched by structure S

4.2.5 defun newDef2Def

[letError p??]
[new2OldDefForm p79]
[new2OldTran p78]

— defun newDef2Def —

```
(defun newDef2Def (def-expr)
 (if (not (and (= (length def-expr) 5) (eq (car def-expr) 'def)))
   (letError "(DEF,form,a,b,c)" def-expr)
   (let ((form (second def-expr))
         (a (third def-expr))
         (b (fourth def-expr))
         (c (fifth def-expr)))
    `(def ,(new2OldDefForm form) ,(new2OldTran a)
          ,(new2OldTran b) ,(new2OldTran c)))))
```

———-

4.2.6 defun new2OldDefForm

[new2OldTran p78]
[new2OldDefForm p79]

— defun new2OldDefForm —

```
(defun new2OldDefForm (x)
 (cond
  ((atom x) (new2OldTran x))
  ((and (listp x) (listp (car x)) (eq (caar x) '|is|) (= (length (car x)) 3))
    (let ((a (second (car x))) (b (third (car x))) (y (cdr x)))
     (new2OldDefForm `((setq ,a ,b) ,@y))))
  ((cons (new2OldTran (car x)) (new2OldDefForm (cdr x))))))
```

———–

4.2.7 defun newConstruct

— defun newConstruct —

```
(defun newConstruct (z)
 (if (atom z)
   z
  `(cons  ,(car z) ,(newConstruct (cdr z)))))
```

———–

4.3 preparse

The first large transformation of this input occurs in the function preparse . The preparse function reads the source file and breaks the input into a list of pairs. The first part of the pair is the line number of the input file and the second part of the pair is the actual source text as a string.

One feature that is the added semicolons at the end of the strings where the "pile" structure of the code has been converted to a semicolon delimited form.

4.3.1 defvar $index

— initvars —

```
(defvar $index 0 "File line number of most recently read line")
```

———–

4.3.2 defvar $linelist

— initvars —

```
(defvar $linelist nil "Stack of preparsed lines")
```

———–

4.3.3 defvar $echolinestack

— initvars —

```
(defvar $echolinestack nil "Stack of lines to list")
```

———-

4.3.4 defvar $preparse-last-line

— initvars —

```
(defvar $preparse-last-line nil "Most recently read line")
```

———-

4.4 Parsing routines

The **initialize-preparse** expects to be called before the **preparse** function. It initializes the state, in particular, it reads a single line from the input stream and stores it in `$preparse-last-line`. The caller gives a stream and the `$preparse-last-line` variable is initialized as:

```
2> (INITIALIZE-PREPARSE #<input stream "/tmp/EQ.spad">)
<2 (INITIALIZE-PREPARSE ")abbrev domain EQ Equation")
```

4.4.1 defun initialize-preparse

[initialize-preparse get-a-line (vol5)]
[$index p80]
[$linelist p80]
[$echolinestack p81]
[$preparse-last-line p81]

— defun initialize-preparse —

```
(defun initialize-preparse (strm)
  (setq $index 0)
  (setq $linelist nil)
  (setq $echolinestack nil)
  (setq $preparse-last-line (get-a-line strm)))
```

———-

The **preparse** function returns a list of pairs of the form: ((linenumber . linestring) .... (linenumber . linestring)) For instance, for the file `EQ.spad`, we get:

```
2> (PREPARSE #<input stream "/tmp/EQ.spad">)
  3> (PREPARSE1 (")abbrev domain EQ Equation"))
```

```
      4> (|doSystemCommand| "abbrev domain EQ Equation")
      <4 (|doSystemCommand| NIL)
    <3 (PREPARSE1 ( ...[snip]... )
   <2 (PREPARSE (
(19 . "Equation(S: Type): public == private where")
(20 . " (Ex ==> OutputForm;")
(21 . "  public ==> Type with")
(22 . "   (\"=\": (S, S) -> $;")
(24 . "    equation: (S, S) -> $;")
(26 . "    swap: $ -> $;")
(28 . "    lhs: $ -> S;")
(30 . "    rhs: $ -> S;")
(32 . "    map: (S -> S, $) -> $;")
(35 . "    if S has InnerEvalable(Symbol,S) then")
(36 . "             InnerEvalable(Symbol,S);")
(37 . "    if S has SetCategory then")
(38 . "       (SetCategory;")
(39 . "        CoercibleTo Boolean;")
(40 . "        if S has Evalable(S) then")
(41 . "          (eval: ($, $) -> $;")
(43 . "           eval: ($, List $) -> $));")
(45 . "    if S has AbelianSemiGroup then")
(46 . "       (AbelianSemiGroup;")
(47 . "        \"+\": (S, $) -> $;")
(50 . "        \"+\": ($, S) -> $);")
(53 . "    if S has AbelianGroup then")
(54 . "       (AbelianGroup;")
(55 . "        leftZero : $ -> $;")
(57 . "        rightZero : $ -> $;")
(59 . "        \"-\": (S, $) -> $;")
(62 . "        \"-\": ($, S) -> $);")
(65 . "    if S has SemiGroup then")
(66 . "       (SemiGroup;")
(67 . "        \"*\": (S, $) -> $;")
(70 . "        \"*\": ($, S) -> $);")
(73 . "    if S has Monoid then")
(74 . "       (Monoid;")
(75 . "        leftOne : $ -> Union($,\"failed\");")
(77 . "        rightOne : $ -> Union($,\"failed\"));")
(79 . "    if S has Group then")
(80 . "       (Group;")
(81 . "        leftOne : $ -> Union($,\"failed\");")
(83 . "        rightOne : $ -> Union($,\"failed\"));")
(85 . "    if S has Ring then")
(86 . "     (Ring;")
(87 . "      BiModule(S,S));")
(88 . "    if S has CommutativeRing then")
(89 . "      Module(S);")
(91 . "    if S has IntegralDomain then")
(92 . "      factorAndSplit : $ -> List $;")
(96 . "    if S has PartialDifferentialRing(Symbol) then")
(97 . "      PartialDifferentialRing(Symbol);")
(98 . "    if S has Field then")
(99 . "     (VectorSpace(S);")
```

```
(100 . "      \"/\": ($, $) -> $;")
(103 . "      inv: $ -> $);")
(105 . "    if S has ExpressionSpace then")
(106 . "        subst: ($, $) -> $);")
(109 . "  private ==> add")
(110 . "   (Rep := Record(lhs: S, rhs: S);")
(111 . "    eq1,eq2: $;")
(112 . "    s : S;")
(113 . "    if S has IntegralDomain then")
(114 . "        factorAndSplit eq ==")
(115 . "         ((S has factor : S -> Factored S) =>")
(116 . "           (eq0 := rightZero eq;")
(117 . "            [equation(rcf.factor,0)
                      for rcf in factors factor lhs eq0]);")
(118 . "          [eq]);")
(119 . "    l:S = r:S      == [l, r];")
(120 . "    equation(l, r) == [l, r];")
(121 . "    lhs eqn        == eqn.lhs;")
(122 . "    rhs eqn        == eqn.rhs;")
(123 . "    swap eqn     == [rhs eqn, lhs eqn];")
(124 . "    map(fn, eqn)   == equation(fn(eqn.lhs), fn(eqn.rhs));")
(125 . "    if S has InnerEvalable(Symbol,S) then")
(126 . "       (s:Symbol;")
(127 . "        ls:List Symbol;")
(128 . "        x:S;")
(129 . "        lx:List S;")
(130 . "        eval(eqn,s,x) == eval(eqn.lhs,s,x) = eval(eqn.rhs,s,x);")
(131 . "        eval(eqn,ls,lx) == eval(eqn.lhs,ls,lx) =
                                    eval(eqn.rhs,ls,lx));")
(132 . "    if S has Evalable(S) then")
(133 . "       (eval(eqn1:$, eqn2:$):$ ==")
(134 . "           eval(eqn1.lhs, eqn2 pretend Equation S) =")
(135 . "               eval(eqn1.rhs, eqn2 pretend Equation S);")
(136 . "        eval(eqn1:$, leqn2:List $):$ ==")
(137 . "           eval(eqn1.lhs, leqn2 pretend List Equation S) =")
(138 . "               eval(eqn1.rhs, leqn2 pretend List Equation S));")
(139 . "    if S has SetCategory then")
(140 . "       (eq1 = eq2 == (eq1.lhs = eq2.lhs)@Boolean and")
(141 . "                     (eq1.rhs = eq2.rhs)@Boolean;")
(142 . "        coerce(eqn:$):Ex == eqn.lhs::Ex = eqn.rhs::Ex;")
(143 . "        coerce(eqn:$):Boolean == eqn.lhs = eqn.rhs);")
(144 . "    if S has AbelianSemiGroup then")
(145 . "       (eq1 + eq2 == eq1.lhs + eq2.lhs = eq1.rhs + eq2.rhs;")
(146 . "        s + eq2 == [s,s] + eq2;")
(147 . "        eq1 + s == eq1 + [s,s]);")
(148 . "    if S has AbelianGroup then")
(149 . "       (- eq == (- lhs eq) = (-rhs eq);")
(150 . "        s - eq2 == [s,s] - eq2;")
(151 . "        eq1 - s == eq1 - [s,s];")
(152 . "        leftZero eq == 0 = rhs eq - lhs eq;")
(153 . "        rightZero eq == lhs eq - rhs eq = 0;")
(154 . "        0 == equation(0$S,0$S);")
(155 . "        eq1 - eq2 == eq1.lhs - eq2.lhs = eq1.rhs - eq2.rhs);")
(156 . "    if S has SemiGroup then")
```

```
(157 . "       (eq1:$ * eq2:$ == eq1.lhs * eq2.lhs = eq1.rhs * eq2.rhs;")
(158 . "        l:S   * eqn:$ == l       * eqn.lhs = l       * eqn.rhs;")
(159 . "        l:S * eqn:$  ==  l * eqn.lhs    =    l * eqn.rhs;")
(160 . "        eqn:$ * l:S  ==  eqn.lhs * l    =    eqn.rhs * l);")
(165 . "    if S has Monoid then")
(166 . "       (1 == equation(1$S,1$S);")
(167 . "        recip eq ==")
(168 . "         ((lh := recip lhs eq) case \"failed\" => \"failed\";")
(169 . "          (rh := recip rhs eq) case \"failed\" => \"failed\";")
(170 . "          [lh :: S, rh :: S]);")
(171 . "        leftOne eq ==")
(172 . "         ((re := recip lhs eq) case \"failed\" => \"failed\";")
(173 . "          1 = rhs eq * re);")
(174 . "        rightOne eq ==")
(175 . "         ((re := recip rhs eq) case \"failed\" => \"failed\";")
(176 . "          lhs eq * re = 1));")
(177 . "    if S has Group then")
(178 . "       (inv eq == [inv lhs eq, inv rhs eq];")
(179 . "        leftOne eq == 1 = rhs eq * inv rhs eq;")
(180 . "        rightOne eq == lhs eq * inv rhs eq = 1);")
(181 . "    if S has Ring then")
(182 . "       (characteristic() == characteristic()$S;")
(183 . "        i:Integer * eq:$ == (i::S) * eq);")
(184 . "    if S has IntegralDomain then")
(185 . "        factorAndSplit eq ==")
(186 . "         ((S has factor : S -> Factored S) =>")
(187 . "           (eq0 := rightZero eq;")
(188 . "            [equation(rcf.factor,0)
                for rcf in factors factor lhs eq0]);")
(189 . "          (S has Polynomial Integer) =>")
(190 . "           (eq0 := rightZero eq;")
(191 . "            MF ==> MultivariateFactorize(Symbol,
                      IndexedExponents Symbol,
                      Integer, Polynomial Integer);")
(193 . "            p : Polynomial Integer :=
                  (lhs eq0) pretend Polynomial Integer;")
(194 . "            [equation((rcf.factor) pretend S,0)
                  for rcf in factors factor(p)$MF]);")
(195 . "          [eq]);")
(196 . "    if S has PartialDifferentialRing(Symbol) then")
(197 . "        differentiate(eq:$, sym:Symbol):$ ==")
(198 . "           [differentiate(lhs eq, sym), differentiate(rhs eq, sym)];")
(199 . "    if S has Field then")
(200 . "       (dimension() == 2 :: CardinalNumber;")
(201 . "        eq1:$ / eq2:$ == eq1.lhs / eq2.lhs = eq1.rhs / eq2.rhs;")
(202 . "        inv eq == [inv lhs eq, inv rhs eq]);")
(203 . "    if S has ExpressionSpace then")
(204 . "        subst(eq1,eq2) ==")
(205 . "           (eq3 := eq2 pretend Equation S;")
(206 . "            [subst(lhs eq1,eq3),subst(rhs eq1,eq3)])))")))
```

4.4.2 defun preparse

[preparse p85]
[preparse1 p85]
[parseprint p500]
[ifcar p??]
[$comblocklist p498]
[$skipme p??]
[$preparse-last-line p81]
[$index p80]
[$docList p??]
[$preparseReportIfTrue p??]
[$headerDocumentation p??]
[$maxSignatureLineNumber p??]
[$constructorLineNumber p??]

— **defun preparse** —

```
(defun preparse (strm &aux (stack ()))
 (declare (special $comblocklist $skipme $preparse-last-line $index |$docList|
           $preparseReportIfTrue |$headerDocumentation|
           |$maxSignatureLineNumber| |$constructorLineNumber|))
  (setq $comblocklist nil)
  (setq $skipme nil)
  (when $preparse-last-line
   (if (consp $preparse-last-line)
    (setq stack $preparse-last-line)
    (push $preparse-last-line stack))
   (setq $index (- $index (length stack))))
  (let ((u (preparse1 stack)))
   (if $skipme
    (preparse strm)
    (progn
      (when $preparseReportIfTrue (parseprint u))
      (setq |$headerDocumentation| nil)
      (setq |$docList| nil)
      (setq |$maxSignatureLineNumber| 0)
      (setq |$constructorLineNumber| (ifcar (ifcar u)))
      u))))
```

———

4.4.3 defun Build the lines from the input for piles

The READLOOP calls preparseReadLine which returns a pair of the form

```
(number . string)
```

[preparseReadLine p??]
[preparse-echo p93]
[fincomblock p498]
[parsepiles p??]
[preparse1 doSystemCommand (vol5)]

[escaped p497]
[indent-pos p498]
[make-full-cvec p??]
[maxindex p??]
[preparse1 strposl (vol5)]
[is-console p499]
[spad-reader p??]
[$echolinestack p81]
[$byConstructors p565]
[$skipme p??]
[$constructorsSeen p565]
[$preparse-last-line p81]
[$preparse-last-line p81]
[$index p80]
[$index p80]
[$linelist p80]
[$in-stream p??]

preparse1 : **(List String)** → **(List (Cons NNI String))** where the input **List String** is the `abbrev` line:

```
(")abbrev domain EQ Equation")
\end{verbatim
and the output is a {\bf List (Cons NNI String)} containing
a line number and the String from that line:
\begin{verbatim}
(... (13 . " (Ex ==> OutputForm;")...)
\end{verbatim
\begin{chunk}{defun preparse1}
(defun preparse1 (linelist)
 (labels (
  (isSystemCommand (line)
    (and (> (length line) 0) (eq (char line 0) #\) )))
  (executeSystemCommand (line)
   (catch 'spad_reader (|doSystemCommand| (subseq line 1))))
 )
 (prog (($linelist linelist) $echolinestack num line i l psloc
        instring pcount comsym strsym oparsym cparsym n ncomsym tmp1
        (sloc -1) continue (parenlev 0) ncomblock lines locs nums functor)
 (declare (special $linelist $echolinestack |$byConstructors| $skipme
           |$constructorsSeen| $preparse-last-line $index in-stream))
READLOOP
  (setq tmp1 (preparseReadLine linelist))
  (setq num (car tmp1))
  (setq line (cdr tmp1))
  (unless (stringp line)
    (preparse-echo linelist)
    (cond
     ((null lines) (return nil))
     (ncomblock    (fincomblock nil nums locs ncomblock nil)))
    (return
     (pair (nreverse nums) (parsepiles (nreverse locs) (nreverse lines)))))
  (when (and (null lines) (isSystemCommand line))
```

```
    (preparse-echo linelist)
    (setq $preparse-last-line nil) ;don't reread this line
    (executeSystemCommand line)
    (go READLOOP))
  (setq l (length line))
  ; if we get a null line, read the next line
  (when (eq l 0) (go READLOOP))
  ; otherwise we have to parse this line
  (setq psloc sloc)
  (setq i 0)
  (setq instring nil)
  (setq pcount 0)
STRLOOP ;; handle things that need ignoring, quoting, or grouping
  ; are we in a comment, quoting, or grouping situation?
  (setq strsym  (or (position #\" line :start  i ) l))
  (setq comsym  (or (search "--"  line :start2 i ) l))
  (setq ncomsym (or (search "++"  line :start2 i ) l))
  (setq oparsym (or (position #\( line :start  i ) l))
  (setq cparsym (or (position #\) line :start  i ) l))
  (setq n (min strsym comsym ncomsym oparsym cparsym))
  (cond
   ; nope, we found no comment, quoting, or grouping
   ((= n l) (go NOCOMS))
   ((escaped line n))
   ; scan until we hit the end of the string
   ((= n strsym) (setq instring (not instring)))
   ; we are in a string, just continue looping
   (instring)
   ;; handle -- comments by ignoring them
   ((= n comsym)
    (setq line (subseq line 0 n))
    (go NOCOMS)) ; discard trailing comment
   ;; handle ++ comments by chunking them together
   ((= n ncomsym)
    (setq sloc (indent-pos line))
    (cond
     ((= sloc n)
      (when (and ncomblock (not (= n (car ncomblock))))
       (fincomblock num nums locs ncomblock linelist)
       (setq ncomblock nil))
      (setq ncomblock (cons n (cons line (ifcdr ncomblock))))
      (setq line ""))
     (t
      (push (strconc (make-full-cvec n " ") (substring line n ())) $linelist)
      (setq $index (1- $index))
      (setq line (subseq line 0 n))))
    (go NOCOMS))
   ; know how deep we are into parens
   ((= n oparsym) (setq pcount (1+ pcount)))
   ((= n cparsym) (setq pcount (1- pcount))))
  (setq i (1+ n))
  (go STRLOOP)
NOCOMS
  ; remember the indentation level
```

```
  (setq sloc (indent-pos line))
  (setq line (string-right-trim " " line))
  (when (null sloc)
   (setq sloc psloc)
   (go READLOOP))
  ; handle line that ends in a continuation character
  (cond
   ((eq (elt line (maxindex line)) #\_)
    (setq continue t)
    (setq line (subseq line (maxindex line))))
   ((setq continue nil)))
  ; test for skipping constructors
  (when (and (null lines) (= sloc 0))
    (if (and |$byConstructors|
             (null (search "==>" line))
             (not
              (member
               (setq functor
                (intern (substring line 0 (strposl ": (=" line 0 nil))))
                |$byConstructors|)))
       (setq $skipme 't)
       (progn
        (push functor |$constructorsSeen|)
        (setq $skipme nil))))
  ; is this thing followed by ++ comments?
  (when (and lines (eql sloc 0))
   (when (and ncomblock (not (zerop (car ncomblock))))
    (fincomblock num nums locs ncomblock linelist))
   (when (not (is-console in-stream))
    (setq $preparse-last-line (nreverse $echolinestack)))
   (return
    (pair (nreverse nums) (parsepiles (nreverse locs) (nreverse lines)))))
  (when (> parenlev 0)
   (push nil locs)
   (setq sloc psloc)
   (go REREAD))
  (when ncomblock
   (fincomblock num nums locs ncomblock linelist)
   (setq ncomblock ()))
  (push sloc locs)
REREAD
  (preparse-echo linelist)
  (push line lines)
  (push num nums)
  (setq parenlev (+ parenlev pcount))
  (when (and (is-console in-stream) (not continue))
   (setq $preparse-last-line nil)
   (return
    (pair (nreverse nums) (parsepiles (nreverse locs) (nreverse lines)))))
  (go READLOOP))))

\end{chunk}

\defun{parsepiles}{parsepiles}
```

```
Add parens and semis to lines to aid parsing.
\calls{parsepiles}{add-parens-and-semis-to-line}
\begin{chunk}{defun parsepiles}
(defun parsepiles (locs lines)
  (mapl #'add-parens-and-semis-to-line
    (nconc lines '(" ")) (nconc locs '(nil)))
  lines)

\end{chunk}

\defun{add-parens-and-semis-to-line}{add-parens-and-semis-to-line}
The line to be worked on is (CAR SLINES).  It's indentation is (CAR SLOCS).
There is a notion of current indentation. Then:
\begin{itemize}
\item Add open paren to beginning of following line if following line's
indentation is greater than current, and add close paren to end of
last succeeding line with following line's indentation.
\item Add semicolon to end of line if following line's indentation is the same.
\item If the entire line consists of the single keyword then or else,
leave it alone."
\end{itemize}
\calls{add-parens-and-semis-to-line}{infixtok}
\calls{add-parens-and-semis-to-line}{drop}
\calls{add-parens-and-semis-to-line}{addclose}
\calls{add-parens-and-semis-to-line}{nonblankloc}
\begin{chunk}{defun add-parens-and-semis-to-line}
(defun add-parens-and-semis-to-line (slines slocs)
 (let ((start-column (car slocs)))
  (when (and start-column (> start-column 0))
   (let ((count 0) (i 0))
    (seq
     (mapl #'(lambda (next-lines nlocs)
              (let ((next-line (car next-lines)) (next-column (car nlocs)))
               (incf i)
               (when next-column
                (setq next-column (abs next-column))
                (when (< next-column start-column) (exit nil))
                (cond
                 ((and (eq next-column start-column)
                       (rplaca nlocs (- (car nlocs)))
                       (not (infixtok next-line)))
                   (setq next-lines (drop (1- i) slines))
                   (rplaca next-lines (addclose (car next-lines) #\;))
                   (setq count (1+ count))))))
                 (cdr slines) (cdr slocs)))
     (when (> count 0)
      (setf (char (car slines) (1- (nonblankloc (car slines)))) #\( )
      (setq slines (drop (1- i) slines))
      (rplaca slines (addclose (car slines) #\) )))))))

\end{chunk}

\defun{preparseReadLine}{preparseReadLine}
\calls{preparseReadLine}{preparseReadLine1}
```

```
\seebook{preparseReadLine}{initial-substring}{5}
\calls{preparseReadLine}{string2BootTree}
\seebook{preparseReadLine}{storeblanks}{5}
\calls{preparseReadLine}{skip-to-endif}
\calls{preparseReadLine}{preparseReadLine}
\refsdollar{preparseReadLine}{*eof*}
\sig{preparseReadLine1}{nil}{(Cons NNI String)}
where the result is a pair with the next input line number (\$index)
and the input string.
\begin{verbatim}
   (0 . ")abbrev domain EQ Equation")
```

— defun preparseReadLine —

```
(defun preparseReadLine (x)
 (let (line ind tmp1)
 (declare (special *eof*))
  (setq tmp1 (preparseReadLine1))
  (setq ind (car tmp1))
  (setq line (cdr tmp1))
  (cond
   ((not (stringp line)) (cons ind line))
   ((zerop (size line))  (cons ind line))
   ((char= (elt line 0) #\) )
    (cond
     ((initial-substring ")if" line)
      (if (eval (string2BootTree (storeblanks line 3)))
       (preparseReadLine x)
       (skip-ifblock x)))
     ((initial-substring ")elseif" line) (skip-to-endif x))
     ((initial-substring ")else" line)   (skip-to-endif x))
     ((initial-substring ")endif" line)  (preparseReadLine x))
     ((initial-substring ")fin" line)
      (setq *eof* t)
      (cons ind nil)))))
  (cons ind line)))
```

———-

4.4.4 defun skip-ifblock

[preparseReadLine1 p91]
[skip-ifblock p90]
[skip-ifblock initial-substring (vol5)]
[string2BootTree p77]
[skip-ifblock storeblanks (vol5)]

— defun skip-ifblock —

```
(defun skip-ifblock (x)
 (let (line ind tmp1)
  (setq tmp1  (preparseReadLine1))
  (setq ind (car tmp1))
```

```
  (setq line (cdr tmp1))
  (cond
   ((not (stringp line))
     (cons ind line))
   ((zerop (size line))
     (skip-ifblock x))
   ((char= (elt line 0) #\) )
    (cond
     ((initial-substring ")if" line)
      (cond
        ((eval (string2BootTree (storeblanks line 3)))
          (preparseReadLine X))
        (t (skip-ifblock x))))
     ((initial-substring ")elseif" line)
       (cond
        ((eval (string2BootTree (storeblanks line 7)))
          (preparseReadLine X))
        (t (skip-ifblock x))))
     ((initial-substring ")else" line)
       (preparseReadLine x))
     ((initial-substring ")endif" line)
       (preparseReadLine x))
     ((initial-substring ")fin" line)
       (cons ind nil))))
   (t (skip-ifblock x)))))
```

———-

4.4.5 defun preparseReadLine1

[preparseReadLine1 get-a-line (vol5)]
[expand-tabs p92]
[maxindex p??]
[strconc p??]
[preparseReadLine1 p91]
[$linelist p80]
[$linelist p80]
[$preparse-last-line p81]
[$index p80]
[$index p80]
[$EchoLineStack p??]

preparseReadLine1 : **nil** → **(Cons NNI String)** where the result is a pair with the next input line number ($index) and the input string.

```
  (0 . ")abbrev domain EQ Equation")
```

— **defun preparseReadLine1** —

```
(defun preparseReadLine1 ()
 (labels (
  (accumulateLinesWithTrailingEscape (line)
```

```
   (let (ind)
   (declare (special $preparse-last-line))
     (if (and (> (setq ind (maxindex line)) -1) (char= (elt line ind) #\_))
      (setq $preparse-last-line
        (strconc (substring line 0 ind) (cdr (preparseReadLine1))))
      line))))
 (let (line)
 (declare (special $linelist $preparse-last-line $index $EchoLineStack))
  (setq line
   (if $linelist
    (pop $linelist)
    (expand-tabs (get-a-line in-stream))))
  (setq $preparse-last-line line)
  (if (stringp line)
   (progn
    (incf $index)   ;; $index is the current line number
    (setq line (string-right-trim " " line))
    (push (copy-seq line) $EchoLineStack)
    (cons $index (accumulateLinesWithTrailingEscape line)))
   (cons $index line)))))
```

———-

4.4.6 defun expand-tabs

[nonblankloc p500]
[indent-pos p498]

— defun expand-tabs —

```
(defun expand-tabs (str)
 (if (and (stringp str) (> (length str) 0))
  (let ((bpos (nonblankloc str))
        (tpos (indent-pos str)))
   (setq str
    (if (eql bpos tpos)
      str
     (concatenate 'string (make-string tpos :initial-element #\space)
                          (subseq str bpos))))
   ;; remove dos CR
   (let ((lpos (maxindex str)))
    (if (eq (char str lpos) #\Return)
      (subseq str 0 lpos)
      str)))
  str))
```

———-

4.5 I/O Handling

4.5.1 defun preparse-echo

[Echo-Meta p??]
[$EchoLineStack p??]

— defun preparse-echo —

```
(defun preparse-echo (linelist)
 (declare (special $EchoLineStack Echo-Meta) (ignore linelist))
 (if Echo-Meta
  (dolist (x (reverse $EchoLineStack))
   (format out-stream "~&;~A~%" x)))
 (setq $EchoLineStack ()))
```

———

4.5.2 Parsing stack

4.5.3 defstruct stack

— initvars —

```
(defstruct stack          "A stack"
           (store nil)    ; contents of the stack
           (size 0)       ; number of elements in Store
           (top nil)      ; first element of Store
           (updated nil)  ; whether something has been pushed on the stack
                          ; since this flag was last set to NIL
)
```

———

4.5.4 defun stack-load

[$stack p93]

— defun stack-load —

```
(defun stack-load (list stack)
  (setf (stack-store stack) list)
  (setf (stack-size stack) (length list))
  (setf (stack-top stack) (car list)))
```

———

4.5.5 defun stack-clear

[$stack p93]

— defun stack-clear —

```
(defun stack-clear (stack)
  (setf (stack-store stack) nil)
  (setf (stack-size stack) 0)
  (setf (stack-top stack) nil)
  (setf (stack-updated stack) nil))
```

———-

4.5.6 defmacro stack-/-empty

[$stack p93]

— defmacro stack-/-empty —

```
(defmacro stack-/-empty (stack) `(> (stack-size ,stack) 0))
```

———-

4.5.7 defun stack-push

[$stack p93]

— defun stack-push —

```
(defun stack-push (x stack)
  (push x (stack-store stack))
  (setf (stack-top stack) x)
  (setf (stack-updated stack) t)
  (incf (stack-size stack))
  x)
```

———-

4.5.8 defun stack-pop

[$stack p93]

— defun stack-pop —

```
(defun stack-pop (stack)
  (let ((y (pop (stack-store stack))))
    (decf (stack-size stack))
    (setf (stack-top stack)
          (if (stack-/-empty stack) (car (stack-store stack))))
    y))
```

———-

4.5.9 Parsing token

4.5.10 defstruct token

A token is a Symbol with a Type. The type is either NUMBER, IDENTIFIER or SPECIAL-CHAR. NonBlank is true if the token is not preceded by a blank.

— initvars —

```
(defstruct token
  (symbol nil)
  (type nil)
  (nonblank t))
```

———-

4.5.11 defvar prior-token

[$token p95]

— initvars —

```
(defvar prior-token (make-token) "What did I see last")
```

———-

4.5.12 defvar nonblank

— initvars —

```
(defvar nonblank t "Is there no blank in front of the current token.")
```

———-

4.5.13 defvar current-token

Token at head of input stream. [$token p95]

— initvars —

```
(defvar current-token (make-token))
```

———-

4.5.14 defvar next-token

[$token p95]

— **initvars** —

```
(defvar next-token (make-token) "Next token in input stream.")
```

———

4.5.15 defvar valid-tokens

[$token p95]

— **initvars** —

```
(defvar valid-tokens 0 "Number of tokens in buffer (0, 1 or 2)")
```

———

4.5.16 defun token-install

[$token p95]

— **defun token-install** —

```
(defun token-install (symbol type token &optional (nonblank t))
  (setf (token-symbol token) symbol)
  (setf (token-type token) type)
  (setf (token-nonblank token) nonblank)
  token)
```

———

4.5.17 defun token-print

[$token p95]

— **defun token-print** —

```
(defun token-print (token)
  (format out-stream "(token (symbol ~S) (type ~S))~%"
          (token-symbol token) (token-type token)))
```

———

4.5.18 Parsing reduction

4.5.19 defstruct reduction

A reduction of a rule is any S-Expression the rule chooses to stack.

— **initvars** —

```
(defstruct (reduction (:type list))
  (rule nil)            ; Name of rule
  (value nil))
```

———

Chapter 5

Parse Transformers

5.1 Direct called parse routines

5.1.1 defun parseTransform

[parseTran p99]
[$defOp p??]

— defun parseTransform —

```
(defun |parseTransform| (x)
 (let (|$defOp|)
 (declare (special |$defOp|))
  (setq |$defOp| nil)
  (setq x (subst '$ '% x :test #'equal)) ; for new compiler compatibility
  (|parseTran| x)))
```

———

5.1.2 defun parseTran

[parseAtom p100]
[parseConstruct p101]
[parseTran p99]
[parseTranList p101]
[getl p??]
[$op p??]

— defun parseTran —

```
(defun |parseTran| (x)
 (labels (
  (g (op)
   (let (tmp1 tmp2 x)
    (seq
```

```
     (if (and (consp op) (eq (qfirst op) '|elt|)
         (progn
          (setq tmp1 (qrest op))
          (and (consp tmp1)
               (progn
                (setq op (qfirst tmp1))
                (setq tmp2 (qrest tmp1))
                (and (consp tmp2)
                     (eq (qrest tmp2) nil)
                         (progn (setq x (qfirst tmp2)) t))))))
        (exit (g x)))
     (exit op)))))
 (let (|$op| argl u r fn)
 (declare (special |$op|))
  (setq |$op| nil)
  (if (atom x)
   (|parseAtom| x)
   (progn
    (setq |$op| (car x))
    (setq argl (cdr x))
    (setq u (g |$op|))
    (cond
     ((eq u '|construct|)
       (setq r (|parseConstruct| argl))
       (if (and (consp |$op|) (eq (qfirst |$op|) '|elt|))
        (cons (|parseTran| |$op|) (cdr r))
        r))
     ((and (atom u) (setq fn (getl u '|parseTran|)))
       (funcall fn argl))
     (t (cons (|parseTran| |$op|) (|parseTranList| argl))))))))
```

———-

5.1.3 defun parseAtom

[parseLeave p123]
[$NoValue p??]

— defun parseAtom —

```
(defun |parseAtom| (x)
 (declare (special |$NoValue|))
 (if (eq x '|break|)
  (|parseLeave| (list '|$NoValue|))
  x))
```

———-

5.1.4 defun parseTranList

[parseTran p99]
[parseTranList p101]

— defun parseTranList —

```
(defun |parseTranList| (x)
 (if (atom x)
  (|parseTran| x)
  (cons (|parseTran| (car x)) (|parseTranList| (cdr x)))))
```

———

5.1.5 defplist parseConstruct

— postvars —

```
(eval-when (eval load)
 (setf (get '|construct| '|parseTran|) '|parseConstruct|))
```

———

5.1.6 defun parseConstruct

[parseTranList p101]
[$insideConstructIfTrue p??]

— defun parseConstruct —

```
(defun |parseConstruct| (u)
 (let (|$insideConstructIfTrue| x)
 (declare (special |$insideConstructIfTrue|))
  (setq |$insideConstructIfTrue| t)
  (setq x (|parseTranList| u))
  (cons '|construct| x)))
```

———

5.2 Indirect called parse routines

In the **parseTran** function there is the code:

```
((and (atom u) (setq fn (getl u '|parseTran|)))
  (funcall fn argl))
```

The functions in this section are called through the symbol-plist of the symbol being parsed. The original list read:

```
  and          parseAnd
```

```
  @           parseAtSign
  CATEGORY    parseCategory
  ::          parseCoerce
  \:          parseColon
  construct   parseConstruct
  DEF         parseDEF
  $<=         parseDollarLessEqual
  $>          parseDollarGreaterThan
  $>=         parseDollarGreaterEqual
  $^=         parseDollarNotEqual
  eqv         parseEquivalence
  exit        parseExit
  >           parseGreaterThan
  >=          parseGreaterEqual
  has         parseHas
  IF          parseIf
  implies     parseImplies
  IN          parseIn
  INBY        parseInBy
  is          parseIs
  isnt        parseIsnt
  Join        parseJoin
  leave       parseLeave
;;control-H   parseLeftArrow
  <=          parseLessEqual
  LET         parseLET
  LETD        parseLETD
  MDEF        parseMDEF
  ^           parseNot
  not         parseNot
  ^=          parseNotEqual
  or          parseOr
  pretend     parsePretend
  return      parseReturn
  SEGMENT     parseSegment
  SEQ         parseSeq
;;control-V   parseUpArrow
  VCONS       parseVCONS
  where       parseWhere
```

5.2.1 defplist parseAnd

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|and| '|parseTran|) '|parseAnd|))
```

————

5.2.2 defun parseAnd

[parseTran p99]
[parseAnd p103]
[parseTranList p101]
[parseIf p117]
[$InteractiveMode p??]

— defun parseAnd —

```
(defun |parseAnd| (arg)
 (cond
  (|$InteractiveMode| (cons '|and| (|parseTranList| arg)))
  ((null arg) '|true|)
  ((null (cdr arg)) (car arg))
  (t
   (|parseIf|
    (list (|parseTran| (car arg)) (|parseAnd| (CDR arg)) '|false| )))))
```

———

5.2.3 defplist parseAtSign

— postvars —

```
(eval-when (eval load)
 (setf (get '@ '|parseTran|) '|parseAtSign|))
```

———

5.2.4 defun parseAtSign

[parseTran p99]
[parseType p104]
[$InteractiveMode p??]

— defun parseAtSign —

```
(defun |parseAtSign| (arg)
 (declare (special |$InteractiveMode|))
 (if |$InteractiveMode|
  (list '@ (|parseTran| (first arg)) (|parseTran| (|parseType| (second arg))))
  (list '@ (|parseTran| (first arg)) (|parseTran| (second arg)))))
```

———

5.2.5 defun parseType

[parseTran p99]

— defun parseType —

```
(defun |parseType| (x)
 (declare (special |$EmptyMode| |$quadSymbol|))
 (setq x (subst |$EmptyMode| |$quadSymbol| x :test #'equal))
 (if (and (consp x) (eq (qfirst x) '|typeOf|)
          (consp (qrest x)) (eq (qcddr x) nil))
    (list '|typeOf| (|parseTran| (qsecond x)))
    x))
```

———-

5.2.6 defplist parseCategory

— postvars —

```
(eval-when (eval load)
 (setf (get 'category '|parseTran|) '|parseCategory|))
```

———-

5.2.7 defun parseCategory

[parseTranList p101]
[parseDropAssertions p104]
[contained p??]

— defun parseCategory —

```
(defun |parseCategory| (arg)
 (let (z key)
  (setq z (|parseTranList| (|parseDropAssertions| arg)))
  (setq key (if (contained '$ z) '|domain| '|package|))
  (cons 'category (cons key z))))
```

———-

5.2.8 defun parseDropAssertions

[parseDropAssertions p104]

— defun parseDropAssertions —

```
(defun |parseDropAssertions| (x)
 (cond
  ((not (consp x)) x)
```

```
  ((and (consp (qfirst x)) (eq (qcaar x) 'if)
        (consp (qcdar x))
        (eq (qcadar x) '|asserted|))
     (|parseDropAssertions| (qrest x)))
  (t (cons (qfirst x) (|parseDropAssertions| (qrest x))))))
```

———-

5.2.9 defplist parseCoerce

— postvars —

```
(eval-when (eval load)
 (setf (get '|::| '|parseTran|) '|parseCoerce|))
```

———-

5.2.10 defun parseCoerce

[parseType p104]
[parseTran p99]
[$InteractiveMode p??]

— defun parseCoerce —

```
(defun |parseCoerce| (arg)
 (if |$InteractiveMode|
  (list '|::|
    (|parseTran| (first arg)) (|parseTran| (|parseType| (second arg))))
  (list '|::| (|parseTran| (first arg)) (|parseTran| (second arg)))))
```

———-

5.2.11 defplist parseColon

— postvars —

```
(eval-when (eval load)
 (setf (get '|:| '|parseTran|) '|parseColon|))
```

———-

5.2.12 defun parseColon

[parseTran p99]
[parseType p104]
[$InteractiveMode p??]

[$insideConstructIfTrue p??]

— defun parseColon —

```
(defun |parseColon| (arg)
 (declare (special |$insideConstructIfTrue|))
 (cond
  ((and (consp arg) (eq (qrest arg) nil))
   (list '|:| (|parseTran| (first arg))))
  ((and (consp arg) (consp (qrest arg)) (eq (qcddr arg) nil))
    (if |$InteractiveMode|
      (if |$insideConstructIfTrue|
        (list 'tag (|parseTran| (first arg))
                   (|parseTran| (second arg)))
        (list '|:| (|parseTran| (first arg))
                   (|parseTran| (|parseType| (second arg)))))
      (list '|:| (|parseTran| (first arg))
                 (|parseTran| (second arg)))))))
```

———-

5.2.13 defplist parseDEF

— postvars —

```
(eval-when (eval load)
 (setf (get 'def '|parseTran|) '|parseDEF|))
```

———-

5.2.14 defun parseDEF

[setDefOp p368]
[parseLhs p107]
[parseTranList p101]
[parseTranCheckForRecord p491]
[opFf p??]
[$lhs p??]

— defun parseDEF —

```
(defun |parseDEF| (arg)
 (let (|$lhs| tList specialList body)
 (declare (special |$lhs|))
  (setq |$lhs| (first arg))
  (setq tList (second arg))
  (setq specialList (third arg))
  (setq body (fourth arg))
  (setDefOp |$lhs|)
  (list 'def (|parseLhs| |$lhs|)
             (|parseTranList| tList)
```

```
      (|parseTranList| specialList)
      (|parseTranCheckForRecord| body (|opOf| |$lhs|)))))
```

———-

5.2.15 defun parseLhs

[parseTran p99]
[transIs p107]

— defun parseLhs —

```
(defun |parseLhs| (x)
 (let (result)
  (cond
   ((atom x) (|parseTran| x))
   ((atom (car x))
     (cons (|parseTran| (car x))
       (dolist (y (cdr x) (nreverse result))
         (push (|transIs| (|parseTran| y)) result))))
   (t (|parseTran| x)))))
```

———-

5.2.16 defun transIs

[isListConstructor p108]
[transIs1 p107]

— defun transIs —

```
(defun |transIs| (u)
  (if (|isListConstructor| u)
    (cons '|construct| (|transIs1| u))
    u))
```

———-

5.2.17 defun transIs1

[nreverse0 p??]
[transIs p107]
[transIs1 p107]

— defun transIs1 —

```
(defun |transIs1| (u)
 (let (x h v tmp3)
  (cond
   ((and (consp u) (eq (qfirst u) '|construct|))
     (dolist (x (qrest u) (nreverse0 tmp3))
```

```
      (push (|transIs| x) tmp3)))
   ((and (consp u) (eq (qfirst u) '|append|) (consp (qrest u))
         (consp (qcddr u)) (eq (qcdddr u) nil))
     (setq x (qsecond u))
     (setq h (list '|:| (|transIs| x)))
     (setq v (|transIs1| (qthird u)))
     (cond
      ((and (consp v) (eq (qfirst v) '|:|)
            (consp (qrest v)) (eq (qcddr v) nil))
         (list h (qsecond v)))
      ((eq v '|nil|) (car (cdr h)))
      ((atom v) (list h (list '|:| v)))
      (t (cons h v))))
   ((and (consp u) (eq (qfirst u) '|cons|) (consp (qrest u))
         (consp (qcddr u)) (eq (qcdddr u) nil))
     (setq h (|transIs| (qsecond u)))
     (setq v (|transIs1| (qthird u)))
     (cond
      ((and (consp v) (eq (qfirst v) '|:|) (consp (qrest v))
            (eq (qcddr v) nil))
         (cons h (list (qsecond v))))
      ((eq v '|nil|) (cons h nil))
      ((atom v) (list h (list '|:| v)))
      (t (cons h v))))
   (t u))))
```

———-

5.2.18 defun isListConstructor

[member p??]

— defun isListConstructor —

```
(defun |isListConstructor| (u)
 (and (consp u) (|member| (qfirst u) '(|construct| |append| |cons|))))
```

———-

5.2.19 defplist parseDollarGreaterthan

— postvars —

```
(eval-when (eval load)
 (setf (get '|$>| '|parseTran|) '|parseDollarGreaterThan|))
```

———-

5.2.20 defun parseDollarGreaterThan

[parseTran p99]
[$op p??]

— defun parseDollarGreaterThan —

```
(defun |parseDollarGreaterThan| (arg)
 (declare (special |$op|))
  (list (subst '$< '$> |$op| :test #'equal)
        (|parseTran| (second arg))
        (|parseTran| (first arg))))
```

———-

5.2.21 defplist parseDollarGreaterEqual

— postvars —

```
(eval-when (eval load)
 (setf (get '|$>=| '|parseTran|) '|parseDollarGreaterEqual|))
```

———-

5.2.22 defun parseDollarGreaterEqual

[parseTran p99]
[$op p??]

— defun parseDollarGreaterEqual —

```
(defun |parseDollarGreaterEqual| (arg)
 (declare (special |$op|))
  (|parseTran| (list '|not| (cons (subst '$< '$>= |$op| :test #'equal) arg))))
```

———-

— postvars —

```
(eval-when (eval load)
 (setf (get '|$<=| '|parseTran|) '|parseDollarLessEqual|))
```

———-

5.2.23 defun parseDollarLessEqual

[parseTran p99]
[$op p??]

— defun parseDollarLessEqual —

```
(defun |parseDollarLessEqual| (arg)
 (declare (special |$op|))
  (|parseTran| (list '|not| (cons (subst '$> '$<= |$op| :test #'equal) arg))))
```

———

5.2.24 defplist parseDollarNotEqual

— postvars —

```
(eval-when (eval load)
 (setf (get '|$^=| '|parseTran|) '|parseDollarNotEqual|))
```

———

5.2.25 defun parseDollarNotEqual

[parseTran p99]
[$op p??]

— defun parseDollarNotEqual —

```
(defun |parseDollarNotEqual| (arg)
 (declare (special |$op|))
 (|parseTran| (list '|not| (cons (subst '$= '$^= |$op| :test #'equal) arg))))
```

———

5.2.26 defplist parseEquivalence

— postvars —

```
(eval-when (eval load)
 (setf (get '|eqv| '|parseTran|) '|parseEquivalence|))
```

———

5.2.27 defun parseEquivalence

[parseIf p117]

— defun parseEquivalence —

```
(defun |parseEquivalence| (arg)
  (|parseIf|
```

```
  (list (first arg) (second arg)
    (|parseIf| (cons (second arg) '(|false| |true|))))))
```

——

5.2.28 defplist parseExit

— postvars —

```
(eval-when (eval load)
 (setf (get '|exit| '|parseTran|) '|parseExit|))
```

——

5.2.29 defun parseExit

[parseTran p99]
[moan p??]

— defun parseExit —

```
(defun |parseExit| (arg)
 (let (a b)
  (setq a (|parseTran| (car arg)))
  (setq b (|parseTran| (cdr arg)))
  (if b
    (cond
     ((null (integerp a))
       (moan "first arg " a " for exit must be integer")
       (list '|exit| 1 a ))
     (t
      (cons '|exit| (cons a b))))
    (list '|exit| 1 a ))))
```

——

5.2.30 defplist parseGreaterEqual

— postvars —

```
(eval-when (eval load)
 (setf (get '|>=| '|parseTran|) '|parseGreaterEqual|))
```

——

5.2.31 defun parseGreaterEqual

[parseTran p99]
[$op p??]

— defun parseGreaterEqual —

```
(defun |parseGreaterEqual| (arg)
 (declare (special |$op|))
 (|parseTran| (list '|not| (cons (subst '< '>= |$op| :test #'equal) arg))))
```

———

5.2.32 defplist parseGreaterThan

— postvars —

```
(eval-when (eval load)
 (setf (get '|>| '|parseTran|) '|parseGreaterThan|))
```

———

5.2.33 defun parseGreaterThan

[parseTran p99]
[$op p??]

— defun parseGreaterThan —

```
(defun |parseGreaterThan| (arg)
 (declare (special |$op|))
 (list (subst '< '> |$op| :test #'equal)
  (|parseTran| (second arg)) (|parseTran| (first arg))))
```

———

5.2.34 defplist parseHas

— postvars —

```
(eval-when (eval load)
 (setf (get '|has| '|parseTran|) '|parseHas|))
```

———

5.2.35 defun parseHas

[unabbrevAndLoad p??]
[getdatabase p??]
[opOf p??]
[makeNonAtomic p??]
[parseHasRhs p114]
[member p??]
[parseType p104]
[nreverse0 p??]
[$InteractiveMode p??]
[$CategoryFrame p??]

— **defun parseHas** —

```
(defun |parseHas| (arg)
 (labels (
  (fn (arg)
   (let (tmp4 tmp6 map op kk)
   (declare (special |$InteractiveMode|))
    (when |$InteractiveMode| (setq arg (|unabbrevAndLoad| arg)))
    (cond
     ((and (consp arg) (eq (qfirst arg) '|:|) (consp (qrest arg))
               (consp (qcddr arg)) (eq (qcdddr arg) nil)
               (consp (qthird arg))
               (eq (qcaaddr arg) '|Mapping|))
        (setq map (rest (third arg)))
        (setq op (second arg))
        (setq op (if (stringp op) (intern op) op))
        (list (list 'signature op map)))
     ((and (consp arg) (eq (qfirst arg) '|Join|))
        (dolist (z (rest arg) tmp4)
         (setq tmp4 (append tmp4 (fn z)))))
     ((and (consp arg) (eq (qfirst arg) 'category))
        (dolist (z (rest arg) tmp6)
          (setq tmp6 (append tmp6 (fn z)))))
     (t
      (setq kk (getdatabase (|opOf| arg) 'constructorkind))
      (cond
       ((or (eq kk '|domain|) (eq kk '|category|))
         (list (|makeNonAtomic| arg)))
       ((and (consp arg) (eq (qfirst arg) 'attribute))
         (list arg))
      ((and (consp arg) (eq (qfirst arg) 'signature))
        (list arg))
      (|$InteractiveMode|
        (|parseHasRhs| arg))
      (t
        (list (list 'attribute arg)))))))))
 (let (tmp1 tmp2 tmp3 x)
 (declare (special |$InteractiveMode| |$CategoryFrame|))
  (setq x (first arg))
  (setq tmp1 (|get| x '|value| |$CategoryFrame|))
```

```
  (when |$InteractiveMode|
   (setq x
     (if (and (consp tmp1) (consp (qrest tmp1)) (consp (qcddr tmp1))
            (eq (qcdddr tmp1) nil)
            (|member| (second tmp1)
             '((|Mode|) (|Domain|) (|SubDomain| (|Domain|)))))
       (first tmp1)
       (|parseType| x))))
  (setq tmp2
   (dolist (u (fn (second arg)) (nreverse0 tmp3))
     (push (list '|has| x u ) tmp3)))
  (if (and (consp tmp2) (eq (qrest tmp2) nil))
   (qfirst tmp2)
   (cons '|and| tmp2)))))
```

———-

5.2.36 defun parseHasRhs

[get p??]
[member p??]
[abbreviation? p??]
[loadLibIfNecessary p114]
[unabbrevAndLoad p??]
[$CategoryFrame p??]

— defun parseHasRhs —

```
(defun |parseHasRhs| (u)
 (let (tmp1 y)
  (declare (special  |$CategoryFrame|))
  (setq tmp1 (|get| u '|value| |$CategoryFrame|))
  (cond
    ((and (consp tmp1) (consp (qrest tmp1))
          (consp (qcddr tmp1)) (eq (qcdddr tmp1) nil)
          (|member| (second tmp1)
             '((|Mode|) (|Domain|) (|SubDomain| (|Domain|)))))
     (second tmp1))
    ((setq y (|abbreviation?| u))
      (if (|loadLibIfNecessary| y t)
        (list (|unabbrevAndLoad| y))
        (list (list 'attribute u))))
    (t (list (list 'attribute u))))))
```

———-

5.2.37 defun loadLibIfNecessary

[loadLibIfNecessary p114]
[canFuncall? p??]
[macrop p??]

[getl p??]
[loadLib p??]
[lassoc p??]
[getProplist p??]
[getdatabase p??]
[updateCategoryFrameForCategory p116]
[updateCategoryFrameForConstructor p115]
[throwKeyedMsg p??]
[$CategoryFrame p??]
[$InteractiveMode p??]

— **defun loadLibIfNecessary** —

```
(defun |loadLibIfNecessary| (u mustExist)
 (let (value y)
 (declare (special |$CategoryFrame| |$InteractiveMode|))
  (cond
   ((eq u '|$EmptyMode|) u)
   ((null (atom u)) (|loadLibIfNecessary| (car u) mustExist))
   (t
    (setq value
     (cond
       ((or (canFuncall? u) (|macrop| u)) u)
       ((getl u 'loaded) u)
       ((|loadLib| u) u)))
    (cond
     ((and (null |$InteractiveMode|)
           (or (null (setq y (|getProplist| u |$CategoryFrame|)))
               (and (null (lassoc '|isFunctor| y))
                    (null (lassoc '|isCategory| y)))))
      (if (setq y (getdatabase u 'constructorkind))
        (if (eq y '|category|)
          (|updateCategoryFrameForCategory| u)
          (|updateCategoryFrameForConstructor| u))
       (|throwKeyedMsg| " %1p is not a known type." (list u))))
     (t value))))))
```

———-

5.2.38 defun updateCategoryFrameForConstructor

[getdatabase p??]
[put p??]
[convertOpAlist2compilerInfo p116]
[addModemap p252]
[$CategoryFrame p??]
[$CategoryFrame p??]

— **defun updateCategoryFrameForConstructor** —

```
(defun |updateCategoryFrameForConstructor| (constructor)
 (let (opAlist tmp1 dc sig pred impl)
```

```
 (declare (special |$CategoryFrame|))
  (setq opalist (getdatabase constructor 'operationalist))
  (setq tmp1 (getdatabase constructor 'constructormodemap))
  (setq dc (caar tmp1))
  (setq sig (cdar tmp1))
  (setq pred (caadr tmp1))
  (setq impl (cadadr tmp1))
  (setq |$CategoryFrame|
   (|put| constructor '|isFunctor|
     (|convertOpAlist2compilerInfo| opAlist)
     (|addModemap| constructor dc sig pred impl
     (|put| constructor '|mode| (cons '|Mapping| sig) |$CategoryFrame|))))))
```

———-

5.2.39 defun convertOpAlist2compilerInfo

— defun convertOpAlist2compilerInfo —

```
(defun |convertOpAlist2compilerInfo| (opalist)
 (labels (
  (formatSig (op arg2)
   (let (typelist slot stuff pred impl)
    (setq typelist (car arg2))
    (setq slot (cadr arg2))
    (setq stuff (cddr arg2))
    (setq pred (if stuff (car stuff) t))
    (setq impl (if (cdr stuff) (cadr stuff) 'elt))
    (list (list op typelist) pred (list impl '$ slot)))))
 (let (data result)
  (setq data
   (loop for item in opalist
    collect
     (loop for sig in (rest item)
      collect (formatSig (car item) sig))))
  (dolist (term data result)
   (setq result (append result term))))))
```

———-

5.2.40 defun updateCategoryFrameForCategory

[getdatabase p??]
[put p??]
[addModemap p252]
[$CategoryFrame p??]
[$CategoryFrame p??]

— defun updateCategoryFrameForCategory —

```
(defun |updateCategoryFrameForCategory| (category)
```

```
(let (tmp1 dc sig pred impl)
 (declare (special |$CategoryFrame|))
  (setq tmp1 (getdatabase category 'constructormodemap))
  (setq dc (caar tmp1))
  (setq sig (cdar tmp1))
  (setq pred (caadr tmp1))
  (setq impl (cadadr tmp1))
  (setq |$CategoryFrame|
    (|put| category '|isCategory| t
     (|addModemap| category dc sig pred impl |$CategoryFrame|)))))
```

———

5.2.41 defplist parseIf

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'if '|parseTran|) '|parseIf|))
```

———

5.2.42 defun parseIf

[parseIf,ifTran p117]
[parseTran p99]

— **defun parseIf** —

```
(defun |parseIf| (arg)
  (if (null (and (consp arg) (consp (qrest arg))
                (consp (qcddr arg)) (eq (qcdddr arg) nil)))
    arg
    (|parseIf,ifTran|
      (|parseTran| (first arg))
      (|parseTran| (second arg))
      (|parseTran| (third arg)))))
```

———

5.2.43 defun parseIf,ifTran

[parseIf,ifTran p117]
[incExitLevel p??]
[makeSimplePredicateOrNil p491]
[incExitLevel p??]
[parseTran p99]
[$InteractiveMode p??]

— **defun parseIf,ifTran** —

```
(defun |parseIf,ifTran| (pred a b)
 (let (pp z ap bp tmp1 tmp2 tmp3 tmp4 tmp5 tmp6 val s)
 (declare (special |$InteractiveMode|))
   (cond
    ((and (null |$InteractiveMode|) (eq pred '|true|))
     a)
    ((and (null |$InteractiveMode|) (eq pred '|false|))
     b)
    ((and (consp pred) (eq (qfirst pred) '|not|)
          (consp (qrest pred)) (eq (qcddr pred) nil))
     (|parseIf,ifTran| (second pred) b a))
    ((and (consp pred) (eq (qfirst pred) 'if)
             (progn
              (setq tmp1 (qrest pred))
              (and (consp tmp1)
                   (progn
                    (setq pp (qfirst tmp1))
                    (setq tmp2 (qrest tmp1))
                    (and (consp tmp2)
                         (progn
                          (setq ap (qfirst tmp2))
                          (setq tmp3 (qrest tmp2))
                          (and (consp tmp3)
                               (eq (qrest tmp3) nil)
                               (progn (setq bp (qfirst tmp3)) t))))))))
     (|parseIf,ifTran| pp
       (|parseIf,ifTran| ap (copy a) (copy b))
       (|parseIf,ifTran| bp a b)))
    ((and (consp pred) (eq (qfirst pred) 'seq)
          (consp (qrest pred)) (progn (setq tmp2 (reverse (qrest pred))) t)
          (and (consp tmp2)
               (consp (qfirst tmp2))
               (eq (qcaar tmp2) '|exit|)
               (progn
                  (setq tmp4 (qcdar tmp2))
                  (and (consp tmp4)
                       (equal (qfirst tmp4) 1)
                       (progn
                        (setq tmp5 (qrest tmp4))
                        (and (consp tmp5)
                             (eq (qrest tmp5) nil)
                             (progn (setq pp (qfirst tmp5)) t)))))
               (progn (setq z (qrest tmp2)) t))
          (progn (setq z (nreverse z)) t))
     (cons 'seq
       (append z
        (list
         (list '|exit| 1 (|parseIf,ifTran| pp
                          (|incExitLevel| a)
                           (|incExitLevel| b)))))))
    ((and (consp a) (eq (qfirst a) 'if) (consp (qrest a))
```

```
          (equal (qsecond a) pred) (consp (qcddr a))
          (consp (qcdddr a))
          (eq (qcddddr a) nil))
     (list 'if pred (third a) b))
    ((and (consp b) (eq (qfirst b) 'if)
          (consp (qrest b)) (equal (qsecond b) pred)
          (consp (qcddr b))
          (consp (qcdddr b))
          (eq (qcddddr b) nil))
     (list 'if pred a (fourth b)))
    ((progn
      (setq tmp1 (|makeSimplePredicateOrNil| pred))
      (and (consp tmp1) (eq (qfirst tmp1) 'seq)
           (progn
            (setq tmp2 (qrest tmp1))
            (and (and (consp tmp2)
                      (progn (setq tmp3 (reverse tmp2)) t))
                 (and (consp tmp3)
                      (progn
                       (setq tmp4 (qfirst tmp3))
                       (and (consp tmp4) (eq (qfirst tmp4) '|exit|)
                            (progn
                             (setq tmp5 (qrest tmp4))
                             (and (consp tmp5) (equal (qfirst tmp5) 1)
                                  (progn
                                   (setq tmp6 (qrest tmp5))
                                   (and (consp tmp6) (eq (qrest tmp6) nil)
                                        (progn (setq val (qfirst tmp6)) t)))))))
                                  (progn (setq s (qrest tmp3)) t))))))
     (setq s (nreverse s))
     (|parseTran|
      (cons 'seq
       (append s
        (list (list '|exit| 1 (|incExitLevel| (list 'if val a b))))))))
    (t
     (list 'if pred a b )))))
```

———-

5.2.44 defplist parseImplies

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|implies| '|parseTran|) '|parseImplies|))
```

———-

5.2.45 defun parseImplies

[parseIf p117]

— defun parseImplies —

```
(defun |parseImplies| (arg)
 (|parseIf| (list (first arg) (second arg) '|true|)))
```

———-

5.2.46 defplist parseIn

— postvars —

```
(eval-when (eval load)
 (setf (get 'in '|parseTran|) '|parseIn|))
```

———-

5.2.47 defun parseIn

[parseTran p99]
[postError p341]

— defun parseIn —

```
(defun |parseIn| (arg)
 (let (i n)
  (setq i (|parseTran| (first arg)))
  (setq n (|parseTran| (second arg)))
  (cond
   ((and (consp n) (eq (qfirst n) 'segment)
         (consp (qrest n)) (eq (qcddr n) nil))
    (list 'step i (second n) 1))
   ((and (consp n) (eq (qfirst n) '|reverse|)
         (consp (qrest n)) (eq (qcddr n) nil)
         (consp (qsecond n)) (eq (qcaadr n) 'segment)
         (consp (qcdadr n))
         (eq (qcddadr n) nil))
    (postError (list "  You cannot reverse an infinite sequence." )))
   ((and (consp n) (eq (qfirst n) 'segment)
         (consp (qrest n)) (consp (qcddr n))
         (eq (qcdddr n) nil))
    (if (third n)
     (list 'step i (second n) 1 (third n))
     (list 'step i (second n) 1)))
   ((and (consp n) (eq (qfirst n) '|reverse|)
         (consp (qrest n)) (eq (qcddr n) nil)
         (consp (qsecond n)) (eq (qcaadr n) 'segment)
         (consp (qcdadr n))
```

```
         (consp (qcddadr n))
         (eq (qrest (qcddadr n)) nil))
    (if (third (second n))
     (list 'step i (third (second n)) -1 (second (second n)))
     (postError (list "  You cannot reverse an infinite sequence."))))
   ((and (consp n) (eq (qfirst n) '|tails|)
         (consp (qrest n)) (eq (qcddr n) nil))
    (list 'on i (second n)))
   (t
    (list 'in i n)))))
```

——–

5.2.48 defplist parseInBy

— postvars —

```
(eval-when (eval load)
 (setf (get 'inby '|parseTran|) '|parseInBy|))
```

——–

5.2.49 defun parseInBy

[postError p341]
[parseTran p99]
[bright p??]
[parseIn p120]

— defun parseInBy —

```
(defun |parseInBy| (arg)
 (let (i n inc u)
  (setq i (first arg))
  (setq n (second arg))
  (setq inc (third arg))
  (setq u (|parseIn| (list i n)))
  (cond
   ((null (and (consp u) (eq (qfirst u) 'step)
               (consp (qrest u))
               (consp (qcddr u))
               (consp (qcdddr u))))
    (postError
     (cons '|   You cannot use|
      (append (|bright| "by")
       (list "except for an explicitly indexed sequence.")))))
   (t
    (setq inc (|parseTran| inc))
    (cons 'step
     (cons (second u)
      (cons (third u)
```

```
      (cons (|parseTran| inc) (cddddr u))))))))
```

———-

5.2.50 defplist parseIs

— postvars —

```
(eval-when (eval load)
 (setf (get '|is| '|parseTran|) '|parseIs|))
```

———-

5.2.51 defun parseIs

[parseTran p99]
[transIs p107]

— defun parseIs —

```
(defun |parseIs| (arg)
 (list '|is| (|parseTran| (first arg)) (|transIs| (|parseTran| (second arg)))))
```

———-

5.2.52 defplist parseIsnt

— postvars —

```
(eval-when (eval load)
 (setf (get '|isnt| '|parseTran|) '|parseIsnt|))
```

———-

5.2.53 defun parseIsnt

[parseTran p99]
[transIs p107]

— defun parseIsnt —

```
(defun |parseIsnt| (arg)
 (list '|isnt|
   (|parseTran| (first arg))
   (|transIs| (|parseTran| (second arg)))))
```

———-

5.2.54 defplist parseJoin

— postvars —

```
(eval-when (eval load)
 (setf (get '|Join| '|parseTran|) '|parseJoin|))
```

———

5.2.55 defun parseJoin

[parseTranList p101]

— defun parseJoin —

```
(defun |parseJoin| (thejoin)
 (labels (
  (fn (arg)
   (cond
    ((null arg)
      nil)
    ((and (consp arg) (consp (qfirst arg)) (eq (qcaar arg) '|Join|))
     (append (cdar arg) (fn (rest arg))))
    (t
     (cons (first arg) (fn (rest arg))))))
 )
 (cons '|Join| (fn (|parseTranList| thejoin)))))
```

———

5.2.56 defplist parseLeave

— postvars —

```
(eval-when (eval load)
 (setf (get '|leave| '|parseTran|) '|parseLeave|))
```

———

5.2.57 defun parseLeave

[parseTran p99]

— defun parseLeave —

```
(defun |parseLeave| (arg)
 (let (a b)
    (setq a (|parseTran| (car arg)))
    (setq b (|parseTran| (cdr arg)))
```

```
    (cond
     (b
      (cond
       ((null (integerp a))
        (moan "first arg " a " for 'leave' must be integer")
        (list '|leave| 1 a))
       (t (cons '|leave| (cons a b)))))
     (t (list '|leave| 1 a)))))
```

———-

5.2.58 defplist parseLessEqual

— postvars —

```
(eval-when (eval load)
 (setf (get '|<=| '|parseTran|) '|parseLessEqual|))
```

———-

5.2.59 defun parseLessEqual

[parseTran p99]
[$op p??]

— defun parseLessEqual —

```
(defun |parseLessEqual| (arg)
 (declare (special |$op|))
 (|parseTran| (list '|not| (cons (subst '> '<= |$op| :test #'equal) arg))))
```

———-

5.2.60 defplist parseLET

— postvars —

```
(eval-when (eval load)
 (setf (get 'let '|parseTran|) '|parseLET|))
```

———-

5.2.61 defun parseLET

[parseTran p99]
[parseTranCheckForRecord p491]
[opOf p??]

[transIs p107]

— defun parseLET —

```
(defun |parseLET| (arg)
 (let (p)
  (setq p
   (list 'let (|parseTran| (first arg))
     (|parseTranCheckForRecord| (second arg) (|opOf| (first arg)))))
  (if (eq (|opOf| (first arg)) '|cons|)
    (list 'let (|transIs| (second p)) (third p))
    p)))
```

———-

5.2.62 defplist parseLETD

— postvars —

```
(eval-when (eval load)
 (setf (get 'letd '|parseTran|) '|parseLETD|))
```

———-

5.2.63 defun parseLETD

[parseTran p99]
[parseType p104]

— defun parseLETD —

```
(defun |parseLETD| (arg)
 (list 'letd
   (|parseTran| (first arg))
   (|parseTran| (|parseType| (second arg)))))
```

———-

5.2.64 defplist parseMDEF

— postvars —

```
(eval-when (eval load)
 (setf (get 'mdef '|parseTran|) '|parseMDEF|))
```

———-

5.2.65 defun parseMDEF

[parseTran p99]
[parseTranList p101]
[parseTranCheckForRecord p491]
[opOf p??]
[$lhs p??]

— defun parseMDEF —

```
(defun |parseMDEF| (arg)
 (let (|$lhs|)
 (declare (special |$lhs|))
  (setq |$lhs| (first arg))
  (list 'mdef
    (|parseTran| |$lhs|)
    (|parseTranList| (second arg))
    (|parseTranList| (third arg))
    (|parseTranCheckForRecord| (fourth arg) (|opOf| |$lhs|)))))
```

———-

5.2.66 defplist parseNot

— postvars —

```
(eval-when (eval load)
 (setf (get '|not| '|parseTran|) '|parseNot|))
```

———-

5.2.67 defplist parseNot

— postvars —

```
(eval-when (eval load)
 (setf (get '|^| '|parseTran|) '|parseNot|))
```

———-

5.2.68 defun parseNot

[parseTran p99]
[$InteractiveMode p??]

— defun parseNot —

```
(defun |parseNot| (arg)
 (declare (special |$InteractiveMode|))
```

```
 (if |$InteractiveMode|
  (list '|not| (|parseTran| (car arg)))
  (|parseTran| (cons 'if (cons (car arg) '(|false| |true|))))))
```

———-

5.2.69 defplist parseNotEqual

— postvars —

```
(eval-when (eval load)
 (setf (get '|^=| '|parseTran|) '|parseNotEqual|))
```

———-

5.2.70 defun parseNotEqual

[parseTran p99]
[$op p??]

— defun parseNotEqual —

```
(defun |parseNotEqual| (arg)
 (declare (special |$op|))
 (|parseTran| (list '|not| (cons (subst '= '^= |$op| :test #'equal) arg))))
```

———-

5.2.71 defplist parseOr

— postvars —

```
(eval-when (eval load)
 (setf (get '|or| '|parseTran|) '|parseOr|))
```

———-

5.2.72 defun parseOr

[parseTran p99]
[parseTranList p101]
[parseIf p117]
[parseOr p127]

— defun parseOr —

```
(defun |parseOr| (arg)
 (let (x)
```

```
  (setq x (|parseTran| (car arg)))
  (cond
   (|$InteractiveMode| (cons '|or| (|parseTranList| arg)))
   ((null arg) '|false|)
   ((null (cdr arg)) (car arg))
   ((and (consp x) (eq (qfirst x) '|not|)
         (consp (qrest x)) (eq (qcddr x) nil))
     (|parseIf| (list (second x) (|parseOr| (cdr arg)) '|true|)))
   (t
     (|parseIf| (list x '|true| (|parseOr| (cdr arg))))))))
```

———-

5.2.73 defplist parsePretend

— postvars —

```
(eval-when (eval load)
 (setf (get '|pretend| '|parseTran|) '|parsePretend|))
```

———-

5.2.74 defun parsePretend

[parseTran p99]
[parseType p104]

— defun parsePretend —

```
(defun |parsePretend| (arg)
 (if |$InteractiveMode|
  (list '|pretend|
        (|parseTran| (first arg))
        (|parseTran| (|parseType| (second arg))))
  (list '|pretend|
         (|parseTran| (first arg))
         (|parseTran| (second arg)))))
```

———-

5.2.75 defplist parseReturn

— postvars —

```
(eval-when (eval load)
 (setf (get '|return| '|parseTran|) '|parseReturn|))
```

———-

5.2.76 defun parseReturn

[parseTran p99]
[moan p??]

— defun parseReturn —

```
(defun |parseReturn| (arg)
 (let (a b)
  (setq a (|parseTran| (car arg)))
  (setq b (|parseTran| (cdr arg)))
  (cond
   (b
    (unless (eql a 1) (moan "multiple-level 'return' not allowed"))
    (cons '|return| (cons 1 b)))
   (t (list '|return| 1 a)))))
```

———

5.2.77 defplist parseSegment

— postvars —

```
(eval-when (eval load)
 (setf (get 'segment '|parseTran|) '|parseSegment|))
```

———

5.2.78 defun parseSegment

[parseTran p99]

— defun parseSegment —

```
(defun |parseSegment| (arg)
  (if (and (consp arg) (consp (qrest arg)) (eq (qcddr arg) nil))
   (if (second arg)
    (list 'segment (|parseTran| (first arg)) (|parseTran| (second arg)))
    (list 'segment (|parseTran| (first arg))))
   (cons 'segment arg)))
```

———

5.2.79 defplist parseSeq

— postvars —

```
(eval-when (eval load)
 (setf (get 'seq '|parseTran|) '|parseSeq|))
```

———-

5.2.80 defun parseSeq

[postError p341]
[transSeq p??]
[mapInto p??]
[last p??]

— defun parseSeq —

```
(defun |parseSeq| (arg)
 (let (tmp1)
  (when (consp arg) (setq tmp1 (reverse arg)))
  (if (null (and (consp arg) (consp tmp1)
                 (consp (qfirst tmp1)) (eq (qcaar tmp1) '|exit|)))
    (postError (list "   Invalid ending to block: " (|last| arg)))
    (|transSeq| (|mapInto| arg '|parseTran|)))))
```

———-

5.2.81 defplist parseVCONS

— postvars —

```
(eval-when (eval load)
 (setf (get 'vcons '|parseTran|) '|parseVCONS|))
```

———-

5.2.82 defun parseVCONS

[parseTranList p101]

— defun parseVCONS —

```
(defun |parseVCONS| (arg)
 (cons 'vector (|parseTranList| arg)))
```

———-

5.2.83 defplist parseWhere

— postvars —

```
(eval-when (eval load)
 (setf (get '|where| '|parseTran|) '|parseWhere|))
```

———-

5.2.84 defun parseWhere

[mapInto p??]

— defun parseWhere —

```
(defun |parseWhere| (arg)
  (cons '|where| (|mapInto| arg '|parseTran|)))
```

———-

Chapter 6

Compile Transformers

With some specific exceptions most compile transformers are invoked through the property list item "`special`". When a specific keyword is encountered in a list form the `compExpression` function looks up the keyword on the property list and funcalls the handler function, passing the form, the mode, and the environment.

If a handler for the keyword is not found then the `compForm` function is called to attempt to compile the form.

6.0.85 defun compExpression

[getl p**??**]
[compForm p541]
[$insideExpressionIfTrue p**??**]

— **defun compExpression** —

```
(defun |compExpression| (form mode env)
 (let (|$insideExpressionIfTrue| fn)
 (declare (special |$insideExpressionIfTrue|))
  (setq |$insideExpressionIfTrue| t)
  (if (and (atom (car form)) (setq fn (getl (car form) 'special)))
    (funcall fn form mode env)
    (|compForm| form mode env))))
```

The functions in this section are called through the symbol-plist of the symbol being parsed. In general, each of these functions takes 3 arguments

1. the **form** which is specific to the function
2. the **mode** a —Join—, which is a set of categories and domains
3. the **env** which is a list of functions and their modemaps

and the functions return modified versions of the three arguments suitable for further processing.

`DEF`	(p137) compDefine
`add`	(p254) compAdd
`@`	(p331) compAtSign
`CAPSULE`	(p256) compCapsule
`case`	(p265) compCase
`Mapping`	(p266) compCat
`Record`	(p266) compCat
`Union`	(p266) compCat
`CATEGORY`	(p267) compCategory
`::`	(p331) compCoerce
`:`	(p271) compColon
`CONS`	(p274) compCons
`construct`	(p276) compConstruct
`ListCategory`	(p277) compConstructorCategory
`RecordCategory`	(p277) compConstructorCategory
`UnionCategory`	(p277) compConstructorCategory
`VectorCategory`	(p277) compConstructorCategory
`elt`	(p285) compElt
`exit`	(p286) compExit
`has`	(p287) compHas(pred mode `$e`)
`IF`	(p289) compIf
`import`	(p296) compImport
`is`	(p296) compIs
`Join`	(p297) compJoin
`+->`	(p299) compLambda
`leave`	(p300) compLeave
`MDEF`	(p300) compMacro
`pretend`	(p301) compPretend
`QUOTE`	(p302) compQuote
`REDUCE`	(p303) compReduce
`COLLECT`	(p305) compRepeatOrCollect
`REPEAT`	(p305) compRepeatOrCollect
`return`	(p307) compReturn
`SEQ`	(p308) compSeq
`LET`	(p311) compSetq
`SETQ`	(p311) compSetq
`String`	(p320) compString
`SubDomain`	(p320) compSubDomain
`SubsetCategory`	(p322) compSubsetCategory
`|`	(p323) compSuchthat
`VECTOR`	(p324) compVector
`where`	(p324) compWhere

6.1 Handline Category DEF forms

This is the graph of the functions used for compDefine. The syntax is a graphviz dot file. To generate this graph as a JPEG file, type:

```
tangle v9compDefine.dot bookvol9.pamphlet >v9compdefine.dot
dot -Tjpg v9compdefine.dot >v9compdefine.jpg
```

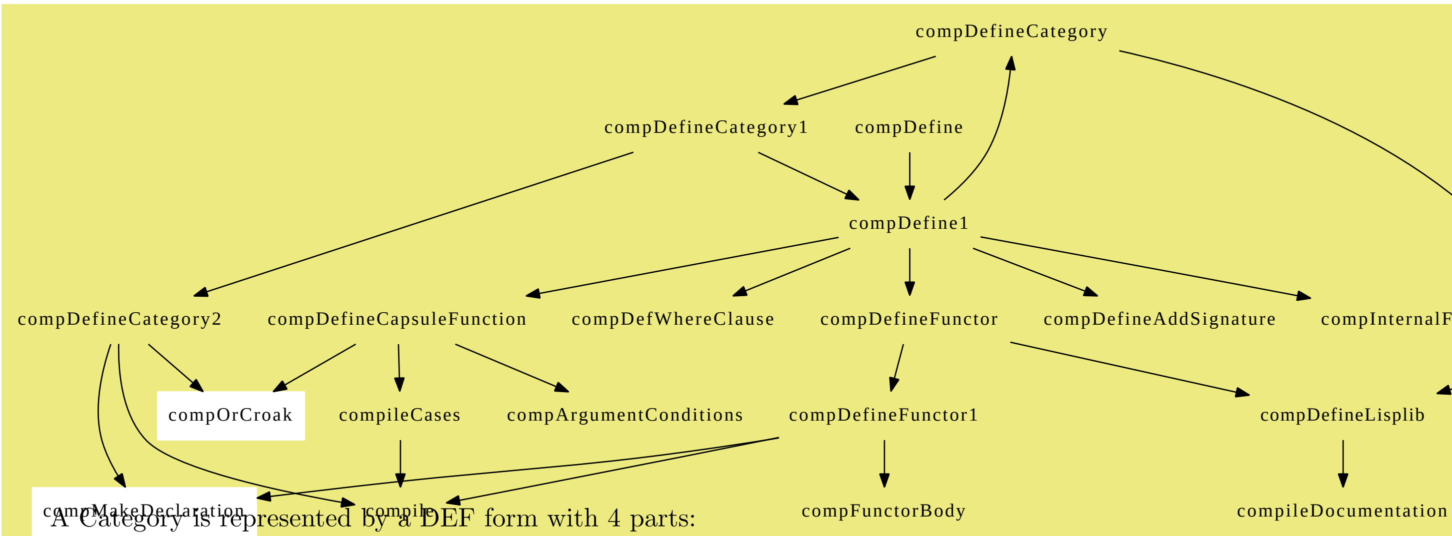
— v9compDefine.dot —

```
digraph pic {
 fontsize=10;
 bgcolor="#ECEA81";
 node [shape=box, color=white, style=filled];

"compArgumentConditions"        [color="#ECEA81"]
"compDefWhereClause"            [color="#ECEA81"]
"compDefine"                    [color="#ECEA81"]
"compDefine1"                   [color="#ECEA81"]
"compDefineAddSignature"        [color="#ECEA81"]
"compDefineCapsuleFunction"     [color="#ECEA81"]
"compDefineCategory"            [color="#ECEA81"]
"compDefineCategory1"           [color="#ECEA81"]
"compDefineCategory2"           [color="#ECEA81"]
"compDefineFunctor"             [color="#ECEA81"]
"compDefineFunctor1"            [color="#ECEA81"]
"compDefineLisplib"             [color="#ECEA81"]
"compInternalFunction"          [color="#ECEA81"]
"compMakeDeclaration"           [color="#FFFFFF"]
"compFunctorBody"               [color="#ECEA81"]
"compOrCroak"                   [color="#FFFFFF"]
"compile"                       [color="#ECEA81"]
"compileCases"                  [color="#ECEA81"]
"compileDocumentation"          [color="#ECEA81"]

"compDefine" -> "compDefine1"
"compDefine1" -> "compDefineCapsuleFunction"
"compDefine1" -> "compDefWhereClause"
"compDefine1" -> "compDefineAddSignature"
"compDefine1" -> "compDefineCategory"
"compDefine1" -> "compDefineFunctor"
"compDefine1" -> "compInternalFunction"
"compDefineCapsuleFunction" -> "compArgumentConditions"
"compDefineCapsuleFunction" -> "compOrCroak"
"compDefineCapsuleFunction" -> "compileCases"
"compDefineCategory" -> "compDefineCategory1"
"compDefineCategory" -> "compDefineLisplib"
"compDefineCategory1" -> "compDefine1"
"compDefineCategory1" -> "compDefineCategory2"
"compDefineCategory2" -> "compMakeDeclaration"
"compDefineCategory2" -> "compOrCroak"
"compDefineCategory2" -> "compile"
"compDefineFunctor" -> "compDefineFunctor1"
"compDefineFunctor" -> "compDefineLisplib"
"compDefineFunctor1" -> "compMakeDeclaration"
"compDefineFunctor1" -> "compFunctorBody"
"compDefineFunctor1" -> "compile"
"compDefineLisplib" -> "compileDocumentation"
"compileCases" -> "compile"
```

```
}
```



A Category is represented by a DEF form with 4 parts:

- a name
- a distnature
- an SC
- a body

For example, the BasicType category is written as

```
BasicType(): Category == with
      "=": (%,%) -> Boolean    ++ x=y tests if x and y are equal.
      "~=": (%,%) -> Boolean   ++ x~=y tests if x and y are not equal.
   add
      _~_=(x:%,y:%) : Boolean == not(x=y)
```

Which compiles to the DEF form:

```
  (DEF
   (|BasicType|)
   ((|Category|))
   (NIL)
   (|add|
     (CATEGORY |domain|
      (SIGNATURE  = ((|Boolean|) $ $))
      (SIGNATURE ~= ((|Boolean|) $ $)))
     (CAPSULE
      (DEF
        (~= |x| |y|)
        ((|Boolean|) $ $)
        (NIL NIL NIL)
        (IF (= |x| |y|) |false| |true|)))))
```

6.1.1 defplist compDefine plist

We set up the `compDefine` function to handle the DEF keyword by setting the `special` keyword on the DEF symbol property list.

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'def 'special) '|compDefine|))
```

———-

6.1.2 defun compDefine

The compDefine function expects three arguments:

1. the **form** which is an def specifying the domain to define.
2. the **mode** a —Join—, which is a set of categories and domains
3. the **env** which is a list of functions and their modemaps

[compDefine1 p137]
[$tripleCache p??]
[$tripleHits p??]
[$macroIfTrue p??]
[$packagesUsed p??]

— **defun compDefine** —

```
(defun |compDefine| (form mode env)
 (let (|$tripleCache| |$tripleHits| |$macroIfTrue| |$packagesUsed|)
 (declare (special |$tripleCache| |$tripleHits| |$macroIfTrue|
                   |$packagesUsed|))
  (setq |$tripleCache| nil)
  (setq |$tripleHits| 0)
  (setq |$macroIfTrue| nil)
  (setq |$packagesUsed| nil)
  (|compDefine1| form mode env)))
```

———-

6.1.3 defun compDefine1

[macroExpand p168]
[isMacro p264]
[getSignatureFromMode p279]
[compDefine1 p137]
[compInternalFunction p150]
[compDefineAddSignature p139]
[compDefWhereClause p151]
[compDefineCategory p153]
[isDomainForm p319]
[getTargetFromRhs p166]

[giveFormalParametersValues p167]
[addEmptyCapsuleIfNecessary p166]
[compDefineFunctor p140]
[stackAndThrow p??]
[strconc p??]
[getAbbreviation p277]
[length p??]
[compDefineCapsuleFunction p147]
[$insideExpressionIfTrue p??]
[$formalArgList p??]
[$form p??]
[$op p??]
[$prefix p??]
[$insideFunctorIfTrue p??]
[$Category p??]
[$insideCategoryIfTrue p??]
[$insideCapsuleFunctionIfTrue p??]
[$ConstructorNames p??]
[$NoValueMode p165]
[$EmptyMode p166]
[$insideWhereIfTrue p??]
[$insideExpressionIfTrue p??]

— defun compDefine1 —

```
(defun |compDefine1| (form mode env)
 (let (|$insideExpressionIfTrue| lhs specialCases sig signature rhs newPrefix
       (tmp1 t))
 (declare (special |$insideExpressionIfTrue| |$formalArgList| |$form|
                   |$op| |$prefix| |$insideFunctorIfTrue| |$Category|
                   |$insideCategoryIfTrue| |$insideCapsuleFunctionIfTrue|
                   |$ConstructorNames| |$NoValueMode| |$EmptyMode|
                   |$insideWhereIfTrue| |$insideExpressionIfTrue|))
  (setq |$insideExpressionIfTrue| nil)
  (setq form (|macroExpand| form env))
  (setq lhs (second form))
  (setq signature (third form))
  (setq specialCases (fourth form))
  (setq rhs (fifth form))
  (cond
   ((and |$insideWhereIfTrue|
         (|isMacro| form env)
         (or (equal mode |$EmptyMode|) (equal mode |$NoValueMode|)))
     (list lhs mode (|put| (car lhs) '|macro| rhs env)))
   ((and (null (car signature)) (consp rhs)
         (null (member (qfirst rhs) |$ConstructorNames|))
         (setq sig (|getSignatureFromMode| lhs env)))
    (|compDefine1|
      (list 'def lhs (cons (car sig) (cdr signature)) specialCases rhs)
      mode env))
   (|$insideCapsuleFunctionIfTrue| (|compInternalFunction| form mode env))
   (t
```

```
    (when (equal (car signature) |$Category|) (setq |$insideCategoryIfTrue| t))
    (setq env (|compDefineAddSignature| lhs signature env))
    (cond
     ((null (dolist (x (rest signature) tmp1) (setq tmp1 (and tmp1 (null x)))))
      (|compDefWhereClause| form mode env))
     ((equal (car signature) |$Category|)
      (|compDefineCategory| form mode env nil |$formalArgList|))
     ((and (|isDomainForm| rhs env) (null |$insideFunctorIfTrue|))
      (when (null (car signature))
        (setq signature
         (cons (|getTargetFromRhs| lhs rhs
                 (|giveFormalParametersValues| (cdr lhs) env))
               (cdr signature))))
      (setq rhs (|addEmptyCapsuleIfNecessary| (car signature) rhs))
      (|compDefineFunctor|
        (list 'def lhs signature specialCases rhs)
        mode env NIL |$formalArgList|))
     ((null |$form|)
      (|stackAndThrow| (list "bad == form " form)))
     (t
      (setq newPrefix
       (if |$prefix|
         (intern (strconc (|encodeItem| |$prefix|) "," (|encodeItem| |$op|)))
         (|getAbbreviation| |$op| (|#| (cdr |$form|)))))
      (|compDefineCapsuleFunction|
         form mode env newPrefix |$formalArgList|)))))))
```

———

6.1.4 defun compDefineAddSignature

[hasFullSignature p166]
[assoc p??]
[lassoc p??]
[getProplist p??]
[comp p530]
[$EmptyMode p166]

— defun compDefineAddSignature —

```
(defun |compDefineAddSignature| (form signature env)
 (let (sig declForm)
 (declare (special |$EmptyMode|))
  (if
   (and (setq sig (|hasFullSignature| (rest form) signature env))
         (null (|assoc| (cons '$ sig)
                        (lassoc '|modemap| (|getProplist| (car form) env)))))
    (progn
     (setq declForm
      (list '|:|
        (cons (car form)
              (loop for x in (rest form)
```

```
                   for m in (rest sig)
               collect (list '|:| x m)))
        (car signature)))
      (third (|comp| declForm |$EmptyMode| env)))
    env)))
```

———-

6.1.5 defun compDefineFunctor

[compDefineLisplib p157]
[compDefineFunctor1 p140]
[$domainShell p??]
[$profileCompiler p??]
[$lisplib p??]
[$profileAlist p??]

— defun compDefineFunctor —

```
(defun |compDefineFunctor| (df mode env prefix fal)
 (let (|$domainShell| |$profileCompiler| |$profileAlist|)
 (declare (special |$domainShell| |$profileCompiler| $lisplib |$profileAlist|))
  (setq |$domainShell| nil)
  (setq |$profileCompiler| t)
  (setq |$profileAlist| nil)
  (if $lisplib
   (|compDefineLisplib| df mode env prefix fal '|compDefineFunctor1|)
   (|compDefineFunctor1| df mode env prefix fal))))
```

———-

6.1.6 defun compDefineFunctor1

[isCategoryPackageName p199]
[getArgumentModeOrMoan p179]
[getModemap p239]
[giveFormalParametersValues p167]
[compMakeCategoryObject p198]
[sayBrightly p??]
[pp p??]
[strconc p??]
[pname p??]
[disallowNilAttribute p204]
[remdup p??]
[NRTgenInitialAttributeAlist p??]
[NRTgetLocalIndex p201]
[compMakeDeclaration p561]
[augModemapsFromCategoryRep p250]
[augModemapsFromCategory p244]
[sublis p??]

[maxindex p??]
[makeFunctorArgumentParameters p206]
[compFunctorBody p162]
[reportOnFunctorCompilation p204]
[compile p163]
[augmentLisplibModemapsFromFunctor p202]
[reportOnFunctorCompilation p204]
[getParentsFor p??]
[computeAncestorsOf p??]
[constructor? p??]
[NRTmakeSlot1Info p??]
[isCategoryPackageName p199]
[lisplibWrite p199]
[mkq p??]
[getdatabase p??]
[NRTgetLookupFunction p200]
[simpBool p??]
[removeZeroOne p??]
[evalAndRwriteLispForm p191]
[$lisplib p??]
[$top-level p??]
[$bootStrapMode p??]
[$CategoryFrame p??]
[$CheckVectorList p??]
[$FormalMapVariableList p249]
[$LocalDomainAlist p??]
[$NRTaddForm p??]
[$NRTaddList p??]
[$NRTattributeAlist p??]
[$NRTbase p??]
[$NRTdeltaLength p??]
[$NRTdeltaListComp p??]
[$NRTdeltaList p??]
[$NRTdomainFormList p??]
[$NRTloadTimeAlist p??]
[$NRTslot1Info p??]
[$NRTslot1PredicateList p??]
[$Representation p??]
[$addForm p??]
[$attributesName p??]
[$byteAddress p??]
[$byteVec p??]
[$compileOnlyCertainItems p??]
[$condAlist p??]
[$domainShell p??]
[$form p??]
[$functionLocations p??]
[$functionStats p??]
[$functorForm p??]

[$functorLocalParameters p??]
[$functorStats p??]
[$functorSpecialCases p??]
[$functorTarget p??]
[$functorsUsed p??]
[$genFVar p??]
[$genSDVar p??]
[$getDomainCode p??]
[$goGetList p??]
[$insideCategoryPackageIfTrue p??]
[$insideFunctorIfTrue p??]
[$isOpPackageName p??]
[$libFile p??]
[$lisplibAbbreviation p??]
[$lisplibAncestors p??]
[$lisplibCategoriesExtended p??]
[$lisplibCategory p??]
[$lisplibForm p??]
[$lisplibKind p??]
[$lisplibMissingFunctions p??]
[$lisplibModemap p??]
[$lisplibOperationAlist p??]
[$lisplibParents p??]
[$lisplibSlot1 p??]
[$lookupFunction p??]
[$myFunctorBody p??]
[$mutableDomain p??]
[$mutableDomains p??]
[$op p??]
[$pairlis p??]
[$QuickCode p??]
[$setelt p??]
[$signature p??]
[$template p??]
[$uncondAlist p??]
[$viewNames p??]
[$lisplibFunctionLocations p??]

— defun compDefineFunctor1 —

```
(defun |compDefineFunctor1| (df mode |$e| |$prefix| |$formalArgList|)
 (declare (special |$e| |$prefix| |$formalArgList|))
 (labels (
  (FindRep (cb)
   (loop while cb do
     (when (atom cb) (return nil))
     (when (and (consp cb) (consp (qfirst cb)) (eq (qcaar cb) 'let)
                (consp (qcdar cb)) (eq (qcadar cb) '|Rep|)
                (consp (qcddar cb)))
      (return (caddar cb)))
      (pop cb))))
```

```
 (let (|$addForm| |$viewNames| |$functionStats| |$functorStats|
          |$form| |$op| |$signature| |$functorTarget|
          |$Representation| |$LocalDomainAlist| |$functorForm|
          |$functorLocalParameters| |$CheckVectorList|
          |$getDomainCode| |$insideFunctorIfTrue| |$functorsUsed|
          |$setelt| $TOP_LEVEL |$genFVar| |$genSDVar|
          |$mutableDomain| |$attributesName| |$goGetList|
          |$condAlist| |$uncondAlist| |$NRTslot1PredicateList|
          |$NRTattributeAlist| |$NRTslot1Info| |$NRTbase|
          |$NRTaddForm| |$NRTdeltaList| |$NRTdeltaListComp|
          |$NRTaddList| |$NRTdeltaLength| |$NRTloadTimeAlist|
          |$NRTdomainFormList| |$template| |$functionLocations|
          |$isOpPackageName| |$lookupFunction| |$byteAddress|
          |$byteVec| form signature body originale argl signaturep target ds
          attributeList parSignature parForm
          argPars opp rettype tt bodyp lamOrSlam fun
          operationAlist modemap libFn tmp1)
(declare (special $lisplib $top_level |$bootStrapMode| |$CategoryFrame|
                |$CheckVectorList| |$FormalMapVariableList|
                |$LocalDomainAlist| |$NRTaddForm| |$NRTaddList|
                |$NRTattributeAlist| |$NRTbase| |$NRTdeltaLength|
                |$NRTdeltaListComp| |$NRTdeltaList| |$NRTdomainFormList|
                |$NRTloadTimeAlist| |$NRTslot1Info| |$NRTslot1PredicateList|
                |$Representation| |$addForm| |$attributesName|
                |$byteAddress| |$byteVec| |$compileOnlyCertainItems|
                |$condAlist| |$domainShell| |$form| |$functionLocations|
                |$functionStats| |$functorForm| |$functorLocalParameters|
                |$functorStats| |$functorSpecialCases| |$functorTarget|
                |$functorsUsed| |$genFVar| |$genSDVar| |$getDomainCode|
                |$goGetList| |$insideCategoryPackageIfTrue|
                |$insideFunctorIfTrue| |$isOpPackageName| |$libFile|
                |$lisplibAbbreviation| |$lisplibAncestors|
                |$lisplibCategoriesExtended| |$lisplibCategory|
                |$lisplibForm| |$lisplibKind| |$lisplibMissingFunctions|
                |$lisplibModemap| |$lisplibOperationAlist| |$lisplibParents|
                |$lisplibSlot1| |$lookupFunction| |$myFunctorBody|
                |$mutableDomain| |$mutableDomains| |$op| |$pairlis|
                |$QuickCode| |$setelt| |$signature| |$template|
                |$uncondAlist| |$viewNames| |$lisplibFunctionLocations|))
 (setq form (second df))
 (setq signature (third df))
 (setq |$functorSpecialCases| (fourth df))
 (setq body (fifth df))
 (setq |$addForm| nil)
 (setq |$viewNames| nil)
 (setq |$functionStats| (list 0 0))
 (setq |$functorStats| (list 0 0))
 (setq |$form| nil)
 (setq |$op| nil)
 (setq |$signature| nil)
 (setq |$functorTarget| nil)
 (setq |$Representation| nil)
 (setq |$LocalDomainAlist| nil)
 (setq |$functorForm| nil)
```

```
(setq |$functorLocalParameters| nil)
(setq |$myFunctorBody| body)
(setq |$CheckVectorList| nil)
(setq |$getDomainCode| nil)
(setq |$insideFunctorIfTrue| t)
(setq |$functorsUsed| nil)
(setq |$setelt| (if  |$QuickCode| 'qsetrefv 'setelt))
(setq $top_level nil)
(setq |$genFVar| 0)
(setq |$genSDVar| 0)
(setq originale |$e|)
(setq |$op| (first form))
(setq argl (rest form))
(setq |$formalArgList| (append argl |$formalArgList|))
(setq |$pairlis|
 (loop for a in argl for v in |$FormalMapVariableList|
  collect (cons a v)))
(setq |$mutableDomain|
                    (OR (|isCategoryPackageName| |$op|)
                        (COND
                          ((boundp '|$mutableDomains|)
                           (member |$op| |$mutableDomains|))
                          ('T NIL))))
(setq signaturep
  (cons (car signature)
        (loop for a in argl collect (|getArgumentModeOrMoan| a form |$e|))))
 (setq |$form| (cons |$op| argl))
 (setq |$functorForm| |$form|)
 (unless (car signaturep)
   (setq signaturep (cdar (|getModemap| |$form| |$e|))))
 (setq target (first signaturep))
 (setq |$functorTarget| target)
 (setq |$e| (|giveFormalParametersValues| argl |$e|))
 (setq tmp1 (|compMakeCategoryObject| target |$e|))
 (if tmp1
  (progn
   (setq ds (first tmp1))
   (setq |$e| (third tmp1))
   (setq |$domainShell| (copy-seq ds))
   (setq |$attributesName| (intern (strconc (pname |$op|) ";attributes")))
   (setq attributeList (|disallowNilAttribute| (elt ds 2)))
   (setq |$goGetList| nil)
   (setq |$condAlist| nil)
   (setq |$uncondAlist| nil)
   (setq |$NRTslot1PredicateList|
    (remdup (loop for x in attributeList collect (second x))))
   (setq |$NRTattributeAlist| (|NRTgenInitialAttributeAlist| attributeList))
   (setq |$NRTslot1Info| nil)
   (setq |$NRTbase| 6)
   (setq |$NRTaddForm| nil)
   (setq |$NRTdeltaList| nil)
   (setq |$NRTdeltaListComp| nil)
   (setq |$NRTaddList| nil)
   (setq |$NRTdeltaLength| 0)
```

```
(setq |$NRTloadTimeAlist| nil)
(setq |$NRTdomainFormList| nil)
(setq |$template| nil)
(setq |$functionLocations| nil)
(loop for x in argl do (|NRTgetLocalIndex| x))
(setq |$e|
  (third (|compMakeDeclaration| (list '|:| '$ target) mode |$e|)))
(unless |$insideCategoryPackageIfTrue|
 (if
   (and (consp body) (eq (qfirst body) '|add|)
        (consp (qrest body))
        (consp (qsecond body))
        (consp (qcddr body))
        (eq (qcdddr body) nil)
        (consp (qthird body))
        (eq (qcaaddr body) 'capsule)
        (member (qcaadr body) '(|List| |Vector|))
        (equal (FindRep (qcdaddr body)) (second body)))
   (setq |$e| (|augModemapsFromCategoryRep| '$
     (second body) (cdaddr body) target |$e|))
   (setq |$e| (|augModemapsFromCategory| '$ '$ target |$e|))))
(setq |$signature| signaturep)
(setq operationAlist (sublis |$pairlis| (elt |$domainShell| 1)))
(setq parSignature (sublis |$pairlis| signaturep))
(setq parForm (sublis |$pairlis| form))
(setq argPars (|makeFunctorArgumentParameters| argl
                (cdr signaturep) (car signaturep)))
(setq |$functorLocalParameters| argl)
(setq opp |$op|)
(setq rettype (CAR signaturep))
(setq tt (|compFunctorBody| body rettype |$e| parForm))
(cond
 (|$compileOnlyCertainItems|
  (|reportOnFunctorCompilation|)
  (list nil (cons '|Mapping| signaturep) originale))
 (t
  (setq bodyp (first tt))
  (setq lamOrSlam (if |$mutableDomain| 'lam 'spadslam))
  (setq fun
   (|compile| (sublis |$pairlis| (list opp (list lamOrSlam argl bodyp)))))
  (setq operationAlist (sublis |$pairlis| |$lisplibOperationAlist|))
  (cond
   ($lisplib
    (|augmentLisplibModemapsFromFunctor| parForm
        operationAlist parSignature)))
  (|reportOnFunctorCompilation|)
  (cond
   ($lisplib
    (setq modemap (list (cons parForm parSignature) (list t opp)))
    (setq |$lisplibModemap| modemap)
    (setq |$lisplibCategory| (cadar modemap))
    (setq |$lisplibParents|
      (|getParentsFor| |$op| |$FormalMapVariableList| |$lisplibCategory|))
    (setq |$lisplibAncestors| (|computeAncestorsOf| |$form| NIL))
```

```
  (setq |$lisplibAbbreviation| (|constructor?| |$op|))))
(setq |$insideFunctorIfTrue| NIL)
(cond
 ($lisplib
  (setq |$lisplibKind|
   (if (and (consp |$functorTarget|)
            (eq (qfirst |$functorTarget|) 'category)
            (consp (qrest |$functorTarget|))
            (not (eq (qsecond |$functorTarget|) '|domain|)))
     '|package|
     '|domain|))
  (setq |$lisplibForm| form)
  (cond
   ((null |$bootStrapMode|)
    (setq |$NRTslot1Info| (|NRTmakeSlot1Info|))
    (setq |$isOpPackageName| (|isCategoryPackageName| |$op|))
    (when |$isOpPackageName|
      (|lisplibWrite| "slot1DataBase"
        (list '|updateSlot1DataBase| (mkq |$NRTslot1Info|))
        |$libFile|))
    (setq |$lisplibFunctionLocations|
       (sublis |$pairlis| |$functionLocations|))
    (setq |$lisplibCategoriesExtended|
       (sublis |$pairlis| |$lisplibCategoriesExtended|))
    (setq libFn (getdatabase opp 'abbreviation))
    (setq |$lookupFunction|
      (|NRTgetLookupFunction| |$functorForm|
        (cadar |$lisplibModemap|) |$NRTaddForm|))
    (setq |$byteAddress| 0)
    (setq |$byteVec| NIL)
    (setq |$NRTslot1PredicateList|
     (loop for x in |$NRTslot1PredicateList|
      collect (|simpBool| x)))
    (|rwriteLispForm| '|loadTimeStuff|
     `(setf (get ,(mkq |$op|) '|infovec|) ,(|getInfovecCode|)))))
  (setq |$lisplibSlot1| |$NRTslot1Info|)
  (setq |$lisplibOperationAlist| operationAlist)
  (setq |$lisplibMissingFunctions| |$CheckVectorList|)))
(|lisplibWrite| "compilerInfo"
 (|removeZeroOne|
  (list 'setq '|$CategoryFrame|
   (list '|put| (list 'quote opp) ''|isFunctor|
          (list 'quote operationAlist)
          (list '|addModemap|
            (list 'quote opp)
            (list 'quote parForm)
            (list 'quote parSignature)
            t
            (list 'quote opp)
            (list '|put| (list 'quote opp) ''|mode|
                   (list 'quote (cons '|Mapping| parSignature))
                   '|$CategoryFrame|)))))
              |$libFile|)
(unless argl
```

```
    (|evalAndRwriteLispForm| 'niladic
      `(setf (get ',opp 'niladic) t)))
   (list fun (cons '|Mapping| signaturep) originale))))
 (progn
 (|sayBrightly| "   cannot produce category object:")
 (|pp| target)
 nil)))))
```

———-

6.1.7 defun compDefineCapsuleFunction

[length p??]
[get p??]
[profileRecord p??]
[compArgumentConditions p160]
[addDomain p233]
[giveFormalParametersValues p167]
[getSignature p282]
[put p??]
[getArgumentModeOrMoan p179]
[checkAndDeclare p283]
[hasSigInTargetCategory p284]
[stripOffSubdomainConditions p281]
[stripOffArgumentConditions p281]
[resolve p334]
[member p??]
[getmode p??]
[formatUnabbreviated p??]
[sayBrightly p??]
[compOrCroak p528]
[NRTassignCapsuleFunctionSlot p??]
[mkq p??]
[replaceExitEtc p309]
[addArgumentConditions p280]
[compileCases p161]
[addStats p??]
[$semanticErrorStack p??]
[$DomainsInScope p??]
[$op p??]
[$formalArgList p??]
[$signatureOfForm p??]
[$functionLocations p??]
[$profileCompiler p??]
[$compileOnlyCertainItems p??]
[$returnMode p??]
[$functorStats p??]
[$functionStats p??]
[$form p??]

[$functionStats p??]
[$argumentConditionList p??]
[$finalEnv p??]
[$initCapsuleErrorCount p??]
[$insideCapsuleFunctionIfTrue p??]
[$CapsuleModemapFrame p??]
[$CapsuleDomainsInScope p??]
[$insideExpressionIfTrue p??]
[$returnMode p??]
[$op p??]
[$formalArgList p??]
[$signatureOfForm p??]
[$functionLocations p??]

— defun compDefineCapsuleFunction —

```
(defun |compDefineCapsuleFunction| (df m oldE |$prefix| |$formalArgList|)
 ; df is ['DEF,form,signature,specialCases,body]
 (declare (special |$prefix| |$formalArgList|))
 (let (|$form| |$op| |$functionStats| |$argumentConditionList| |$finalEnv|
       |$initCapsuleErrorCount| |$insideCapsuleFunctionIfTrue|
       |$CapsuleModemapFrame| |$CapsuleDomainsInScope|
       |$insideExpressionIfTrue| form signature body tmp1 lineNumber
       specialCases argl identSig argModeList signaturep e rettype tmp2
       localOrExported formattedSig tt catchTag bodyp finalBody fun val)
 (declare (special |$form| |$op| |$functionStats| |$functorStats|
                   |$argumentConditionList| |$finalEnv| |$returnMode|
                   |$initCapsuleErrorCount| |$newCompCompare| |$NoValueMode|
                   |$insideCapsuleFunctionIfTrue|
                   |$CapsuleModemapFrame| |$CapsuleDomainsInScope|
                   |$insideExpressionIfTrue| |$compileOnlyCertainItems|
                   |$profileCompiler| |$functionLocations| |$finalEnv|
                   |$signatureOfForm| |$semanticErrorStack|))
  (setq form (second df))
  (setq signature (third df))
  (setq specialCases (fourth df))
  (setq body (fifth df))
  (setq tmp1 specialCases)
  (setq lineNumber (first tmp1))
  (setq specialCases (rest tmp1))
  (setq e oldE)
;-1. bind global variables
  (setq |$form| nil)
  (setq |$op| nil)
  (setq |$functionStats| (list 0 0))
  (setq |$argumentConditionList| nil)
  (setq |$finalEnv| nil)
; used by ReplaceExitEtc to get a common environment
  (setq |$initCapsuleErrorCount| (|#| |$semanticErrorStack|))
  (setq |$insideCapsuleFunctionIfTrue| t)
  (setq |$CapsuleModemapFrame| e)
  (setq |$CapsuleDomainsInScope| (|get| '|$DomainsInScope| 'special e))
  (setq |$insideExpressionIfTrue| t)
```

```
  (setq |$returnMode| m)
  (setq |$op| (first form))
  (setq argl (rest form))
  (setq |$form| (cons |$op| argl))
  (setq argl (|stripOffArgumentConditions| argl))
  (setq |$formalArgList| (append argl |$formalArgList|))
; let target and local signatures help determine modes of arguments
  (setq argModeList
   (cond
    ((setq identSig (|hasSigInTargetCategory| argl form (car signature) e))
      (setq e (|checkAndDeclare| argl form identSig e))
      (cdr identSig))
    (t
     (loop for a in argl
      collect (|getArgumentModeOrMoan| a form e)))))
  (setq argModeList (|stripOffSubdomainConditions| argModeList argl))
  (setq signaturep (cons (car signature) argModeList))
  (unless identSig
    (setq oldE (|put| |$op| '|mode| (cons '|Mapping| signaturep) oldE)))
; obtain target type if not given
  (cond
   ((null (car signaturep))
     (setq signaturep
      (cond
       (identSig identSig)
       (t (|getSignature| |$op| (cdr signaturep) e))))))
  (when signaturep
   (setq e (|giveFormalParametersValues| argl e))
   (setq |$signatureOfForm| signaturep)
   (setq |$functionLocations|
     (cons (cons (list |$op| |$signatureOfForm|) lineNumber)
           |$functionLocations|))
   (setq e (|addDomain| (car signaturep) e))
   (setq e (|compArgumentConditions| e))
   (when |$profileCompiler|
    (loop for x in argl for y in signaturep
     do (|profileRecord| '|arguments| x y)))
; 4. introduce needed domains into extendedEnv
   (loop for domain in signaturep
    do (setq e (|addDomain| domain e)))
; 6. compile body in environment with extended environment
   (setq rettype (|resolve| (car signaturep) |$returnMode|))
   (setq localOrExported
    (if (and (null (|member| |$op| |$formalArgList|))
             (eq (first tmp2) '|Mapping|))
       '|local|
       '|exported|))
; 6a skip if compiling only certain items but not this one
; could be moved closer to the top
   (setq formattedSig (|formatUnabbreviated| (cons '|Mapping| signaturep)))
   (cond
    ((and |$compileOnlyCertainItems|
          (null (|member| |$op| |$compileOnlyCertainItems|)))
     (|sayBrightly|
```

```
      (cons "   skipping " (cons localOrExported (|bright| |$op|))))
     (list nil (cons '|Mapping| signaturep) oldE))
    (t
     (|sayBrightly|
      (cons "   compiling " (cons localOrExported (append (|bright| |$op|)
         (cons ": " formattedSig)))))
     (setq tt (catch '|compCapsuleBody| (|compOrCroak| body rettype e)))
     (|NRTassignCapsuleFunctionSlot| |$op| signaturep)
; A THROW to the above CATCH occurs if too many semantic errors occur
; see stackSemanticError
     (setq catchTag (mkq (gensym)))
     (setq fun
      (progn
       (setq bodyp
        (|replaceExitEtc| (car tt) catchTag '|TAGGEDreturn| |$returnMode|))
       (setq bodyp (|addArgumentConditions| bodyp |$op|))
       (setq finalBody (list 'catch catchTag bodyp))
       (|compileCases|
         (list |$op| (list 'lam (append argl (list '$)) finalBody))
         oldE)))
     (setq |$functorStats| (|addStats| |$functorStats| |$functionStats|))
; 7. give operator a 'value property
     (setq val (list fun signaturep e))
     (list fun (list '|Mapping| signaturep) oldE))))))
```

———-

6.1.8 defun compInternalFunction

[identp p??]
[stackAndThrow p??]

— defun compInternalFunction —

```
(defun |compInternalFunction| (df m env)
 (let (form signature specialCases body op argl nbody nf ress)
  (setq form (second df))
  (setq signature (third df))
  (setq specialCases (fourth df))
  (setq body (fifth df))
  (setq op (first form))
  (setq argl (rest form))
  (cond
   ((null (identp op))
     (|stackAndThrow| (list '|Bad name for internal function:| op)))
   ((eql (|#| argl) 0)
     (|stackAndThrow|
      (list '|Argumentless internal functions unsupported:| op )))
   (t
    (setq nbody (list '+-> argl body))
    (setq nf (list 'let (list '|:| op (cons '|Mapping| signature)) nbody))
    (setq ress (|comp| nf m env)) ress))))
```

———-

6.1.9 defun compDefWhereClause

[getmode p??]
[userError p??]
[concat p??]
[lassoc p??]
[pairList p??]
[union p??]
[listOfIdentifersIn p??]
[delete p??]
[orderByDependency p211]
[assocleft p??]
[assocright p??]
[comp p530]
[$sigAlist p??]
[$predAlist p??]

— defun compDefWhereClause —

```
(defun |compDefWhereClause| (arg mode env)
 (labels (
  (transformType (x)
   (declare (special |$sigAlist|))
   (cond
    ((atom x) x)
    ((and (consp x) (eq (qfirst x) '|:|) (consp (qrest x))
          (consp (qcddr x)) (eq (qcdddr x) nil))
     (setq |$sigAlist|
      (cons (cons (second x) (transformType (third x)))
      |$sigAlist|))
     x)
   ((and (consp x) (eq (qfirst x) '|Record|)) x)
   (t
    (cons (first x)
     (loop for y in (rest x)
      collect (transformType y))))))
  (removeSuchthat (x)
   (declare (special |$predAlist|))
    (if (and (consp x) (eq (qfirst x) '|\||) (consp (qrest x))
             (consp (qcddr x)) (eq (qcdddr x) nil))
     (progn
      (setq |$predAlist| (cons (cons (second x) (third x)) |$predAlist|))
      (second x))
     x))
  (fetchType (a x env form)
   (if x
    x
    (or (|getmode| a env)
        (|userError| (|concat|
         "There is no mode for argument" a "of function" (first form))))))
```

```
  (addSuchthat (x y)
   (let (p)
   (declare (special |$predAlist|))
     (if (setq p (lassoc x |$predAlist|)) (list '|\|| y p) y)))
 )
 (let (|$sigAlist| |$predAlist| form signature specialCases body sigList
       argList argSigAlist argDepAlist varList whereList formxx signaturex
       defform formx)
 (declare (special |$sigAlist| |$predAlist|))
; form is lhs (f a1 ... an) of definition; body is rhs;
; signature is (t0 t1 ... tn) where t0= target type, ti=type of ai, i > 0;
; specialCases is (NIL l1 ... ln) where li is list of special cases
; which can be given for each ti
;
; removes declarative and assignment information from form and
; signature, placing it in list L, replacing form by ("where",form',:L),
; signature by a list of NILs (signifying declarations are in e)
  (setq form (second arg))
  (setq signature (third arg))
  (setq specialCases (fourth arg))
  (setq body (fifth arg))
  (setq |$sigAlist| nil)
  (setq |$predAlist| nil)
; 1. create sigList= list of all signatures which have embedded
;    declarations moved into global variable $sigAlist
  (setq sigList
   (loop for a in (rest form) for x in (rest signature)
    collect (transformType (fetchType a x env form))))
; 2. replace each argument of the form (|| x p) by x, recording
;    the given predicate in global variable $predAlist
  (setq argList
   (loop for a in (rest form)
    collect (removeSuchthat a)))
  (setq argSigAlist (append |$sigAlist| (pairList argList sigList)))
  (setq argDepAlist
   (loop for pear in argSigAlist
    collect
     (cons (car pear)
      (|union| (|listOfIdentifiersIn| (cdr pear))
       (|delete| (car pear)
                 (|listOfIdentifiersIn| (lassoc (car pear) |$predAlist|)))))))
; 3. obtain a list of parameter identifiers (x1 .. xn) ordered so that
;       the type of xi is independent of xj if i < j
  (setq varList
   (|orderByDependency| (assocleft argDepAlist) (assocright argDepAlist)))
; 4. construct a WhereList which declares and/or defines the xi's in
;    the order constructed in step 3
  (setq whereList
   (loop for x in varList
    collect (addSuchthat x (list '|:| x (lassoc x argSigAlist)))))
  (setq formxx (cons (car form) argList))
  (setq signaturex
   (cons (car signature)
    (loop for x in (rest signature) collect nil)))
```

```
  (setq defform (list 'def formxx signaturex specialCases body))
  (setq formx (cons '|where| (cons defform whereList)))
; 5. compile new ('DEF,("where",form',:WhereList),:.) where
;    all argument parameters of form' are bound/declared in WhereList
  (|comp| formx mode env))))
```

———-

6.1.10 defun compDefineCategory

[compDefineLisplib p157]
[compDefineCategory1 p153]
[$domainShell p??]
[$lisplibCategory p??]
[$lisplib p??]
[$insideFunctorIfTrue p??]

— defun compDefineCategory —

```
(defun |compDefineCategory| (df mode env prefix fal)
 (let (|$domainShell| |$lisplibCategory|)
 (declare (special |$domainShell| |$lisplibCategory| $lisplib
                   |$insideFunctorIfTrue|))
  (setq |$domainShell| nil) ; holds the category of the object being compiled
  (setq |$lisplibCategory| nil)
  (if (and (null |$insideFunctorIfTrue|) $lisplib)
    (|compDefineLisplib| df mode env prefix fal '|compDefineCategory1|)
    (|compDefineCategory1| df mode env prefix fal))))
```

———-

6.1.11 defun compDefineCategory1

[compDefineCategory2 p154]
[makeCategoryPredicates p169]
[compDefine1 p137]
[mkCategoryPackage p169]
[$insideCategoryPackageIfTrue p??]
[$EmptyMode p166]
[$categoryPredicateList p??]
[$lisplibCategory p??]
[$bootStrapMode p??]

— defun compDefineCategory1 —

```
(defun |compDefineCategory1| (df mode env prefix fal)
 (let (|$insideCategoryPackageIfTrue| |$categoryPredicateList| form
       sig sc cat body categoryCapsule d tmp1 tmp3)
 (declare (special |$insideCategoryPackageIfTrue| |$EmptyMode|
                   |$categoryPredicateList| |$lisplibCategory|
                   |$bootStrapMode|))
```

```
;; a category is a DEF form with 4 parts:
;; ((DEF (|BasicType|) ((|Category|)) (NIL)
;;    (|add| (CATEGORY |domain| (SIGNATURE = ((|Boolean|) $ $))
;;               (SIGNATURE ~= ((|Boolean|) $ $)))
;;          (CAPSULE (DEF (~= |x| |y|) ((|Boolean|) $ $) (NIL NIL NIL)
;;                         (IF (= |x| |y|) |false| |true|))))))
(setq form (second df))
(setq sig (third df))
(setq sc (fourth df))
(setq body (fifth df))
(setq categoryCapsule
 (when (and (consp body) (eq (qfirst body) '|add|)
           (consp (qrest body)) (consp (qcddr body))
           (eq (qcdddr body) nil))
   (setq tmp1 (third body))
   (setq body (second body))
   tmp1))
(setq tmp3 (|compDefineCategory2| form sig sc body mode env prefix fal))
(setq d (first tmp3))
(setq mode (second tmp3))
(setq env (third tmp3))
(when (and categoryCapsule (null |$bootStrapMode|))
  (setq |$insideCategoryPackageIfTrue| t)
  (setq |$categoryPredicateList|
     (|makeCategoryPredicates| form |$lisplibCategory|))
  (setq env (third
   (|compDefine1|
     (|mkCategoryPackage| form cat categoryCapsule) |$EmptyMode| env))))
(list d mode env)))
```

———

6.1.12 defun compDefineCategory2

[addBinding p??]
[getArgumentModeOrMoan p179]
[giveFormalParametersValues p167]
[take p??]
[sublis p??]
[compMakeDeclaration p561]
[opOf p??]
[optFunctorBody p??]
[compOrCroak p528]
[mkConstructor p191]
[compile p163]
[lisplibWrite p199]
[removeZeroOne p??]
[mkq p??]
[evalAndRwriteLispForm p191]
[eval p??]
[getParentsFor p??]

[computeAncestorsOf p??]
[constructor? p??]
[augLisplibModemapsFromCategory p180]
[$prefix p??]
[$formalArgList p??]
[$definition p??]
[$form p??]
[$op p??]
[$extraParms p??]
[$lisplibCategory p??]
[$FormalMapVariableList p249]
[$libFile p??]
[$TriangleVariableList p??]
[$lisplib p??]
[$formalArgList p??]
[$insideCategoryIfTrue p??]
[$top-level p??]
[$definition p??]
[$form p??]
[$op p??]
[$extraParms p??]
[$functionStats p??]
[$functorStats p??]
[$frontier p??]
[$getDomainCode p??]
[$addForm p??]
[$lisplibAbbreviation p??]
[$functorForm p??]
[$lisplibAncestors p??]
[$lisplibCategory p??]
[$lisplibParents p??]
[$lisplibModemap p??]
[$lisplibKind p??]
[$lisplibForm p??]
[$domainShell p??]

— **defun compDefineCategory2** —

```
(defun |compDefineCategory2|
       (form signature specialCases body mode env |$prefix| |$formalArgList|)
 (declare (special |$prefix| |$formalArgList|) (ignore specialCases))
 (let (|$insideCategoryIfTrue| $TOP_LEVEL |$definition| |$form| |$op|
       |$extraParms| |$functionStats| |$functorStats| |$frontier|
       |$getDomainCode| |$addForm| argl sargl aList signaturep opp  formp
       formalBody formals actuals g fun pairlis parSignature parForm modemap)
 (declare (special |$insideCategoryIfTrue| $top_level |$definition|
                   |$form| |$op| |$extraParms| |$functionStats|
                   |$functorStats| |$frontier| |$getDomainCode|
                   |$addForm| |$lisplibAbbreviation| |$functorForm|
                   |$lisplibAncestors| |$lisplibCategory|
                   |$FormalMapVariableList| |$lisplibParents|
```

```
                    |$lisplibModemap| |$lisplibKind| |$lisplibForm|
                    $lisplib |$domainShell| |$libFile|
                    |$TriangleVariableList|))
; 1. bind global variables
  (setq |$insideCategoryIfTrue| t)
  (setq $top_level nil)
  (setq |$definition| nil)
  (setq |$form| nil)
  (setq |$op| nil)
  (setq |$extraParms| nil)
; 1.1  augment e to add declaration $: <form>
  (setq |$definition| form)
  (setq |$op| (car |$definition|))
  (setq argl (cdr |$definition|))
  (setq env (|addBinding| '$  (list (cons '|mode| |$definition|)) env))
; 2. obtain signature
  (setq signaturep
    (cons (car signature)
     (loop for a in argl
      collect (|getArgumentModeOrMoan| a |$definition| env))))
  (setq env (|giveFormalParametersValues| argl env))
; 3. replace arguments by $1,..., substitute into body,
;    and introduce declarations into environment
  (setq sargl (take (|#| argl) |$TriangleVariableList|))
  (setq |$form| (cons |$op| sargl))
  (setq |$functorForm| |$form|)
  (setq |$formalArgList| (append sargl |$formalArgList|))
  (setq aList (loop for a in argl for sa in sargl collect (cons a sa)))
  (setq formalBody (sublis aList body))
  (setq signaturep (sublis aList signaturep))
  ; Begin lines for category default definitions
  (setq |$functionStats| (list 0 0))
  (setq |$functorStats| (list 0 0))
  (setq |$frontier| 0)
  (setq |$getDomainCode| nil)
  (setq |$addForm| nil)
  (loop for x in sargl for r in (rest signaturep)
   do (setq env (third (|compMakeDeclaration| (list '|:| x r) mode env))))
; 4. compile body in environment of %type declarations for arguments
  (setq opp |$op|)
  (when (and (not (eq (|opOf| formalBody) '|Join|))
             (not (eq (|opOf| formalBody) '|mkCategory|)))
    (setq formalBody (list '|Join| formalBody)))
  (setq body
    (|optFunctorBody| (car (|compOrCroak| formalBody (car signaturep) env))))
  (when |$extraParms|
    (setq actuals nil)
    (setq formals nil)
    (loop for u in |$extraParms| do
      (setq formals (cons (car u) formals))
      (setq actuals (cons (mkq (cdr u)) actuals)))
    (setq body
     (list '|sublisV| (list 'pair (list 'quote formals) (cons 'list actuals))
           body)))
```

```
;  always subst for args after extraparms
  (when argl
    (setq body
     (list '|sublisV|
      (list 'pair
       (list 'quote sargl)
        (cons 'list (loop for u in sargl collect (list '|devaluate| u))))
        body)))
  (setq body
   (list 'prog1 (list 'let (setq g (gensym)) body)
               (list 'setelt g 0 (|mkConstructor| |$form|))))
  (setq fun (|compile| (list opp (list 'lam sargl body))))
; 5. give operator a 'modemap property
  (setq pairlis
   (loop for a in argl for v in |$FormalMapVariableList|
    collect (cons a v)))
  (setq parSignature (sublis pairlis signaturep))
  (setq parForm (sublis pairlis form))
  (|lisplibWrite| "compilerInfo"
    (|removeZeroOne|
     (list 'setq '|$CategoryFrame|
       (list '|put| (list 'quote opp) ''|isCategory| t
              (list '|addModemap| (mkq opp) (mkq parForm)
                     (mkq parSignature) t (mkq fun) '|$CategoryFrame|))))
    |$libFile|)
  (unless sargl
   (|evalAndRwriteLispForm| 'niladic
    `(setf (get ',opp 'niladic) t)))
;; 6 put modemaps into InteractiveModemapFrame
 (setq |$domainShell| (|eval| (cons opp (mapcar 'mkq sargl))))
 (setq |$lisplibCategory| formalBody)
 (when $lisplib
   (setq |$lisplibForm| form)
   (setq |$lisplibKind| '|category|)
   (setq modemap (list (cons parForm parSignature) (list t opp)))
   (setq |$lisplibModemap| modemap)
   (setq |$lisplibParents|
     (|getParentsFor| |$op| |$FormalMapVariableList| |$lisplibCategory|))
   (setq |$lisplibAncestors| (|computeAncestorsOf| |$form| nil))
   (setq |$lisplibAbbreviation| (|constructor?| |$op|))
   (setq formp (cons opp sargl))
   (|augLisplibModemapsFromCategory| formp formalBody signaturep))
 (list fun '(|Category|) env)))
```

———

6.1.13 defun compDefineLisplib

[sayMSG p??]
[fillerSpaces p??]
[getConstructorAbbreviation p??]
[compileDocumentation p160]

[bright p??]
[finalizeLisplib p194]
[rshut p??]
[lisplibDoRename p192]
[filep p??]
[rpackfile p??]
[unloadOneConstructor p192]
[localdatabase p??]
[getdatabase p??]
[updateCategoryFrameForCategory p116]
[updateCategoryFrameForConstructor p115]
[$compileDocumentation p160]
[$filep p??]
[$spadLibFT p??]
[$algebraOutputStream p??]
[$newConlist p501]
[$lisplibKind p??]
[$lisplib p??]
[$op p??]
[$lisplibParents p??]
[$lisplibPredicates p??]
[$lisplibCategoriesExtended p??]
[$lisplibForm p??]
[$lisplibKind p??]
[$lisplibAbbreviation p??]
[$lisplibAncestors p??]
[$lisplibModemap p??]
[$lisplibModemapAlist p??]
[$lisplibSlot1 p??]
[$lisplibOperationAlist p??]
[$lisplibSuperDomain p??]
[$libFile p??]
[$lisplibVariableAlist p??]
[$lisplibCategory p??]
[$newConlist p501]

— defun compDefineLisplib —

```
(defun |compDefineLisplib| (df m env prefix fal fn)
 (let ($LISPLIB |$op| |$lisplibAttributes| |$lisplibPredicates|
       |$lisplibCategoriesExtended| |$lisplibForm| |$lisplibKind|
       |$lisplibAbbreviation| |$lisplibParents| |$lisplibAncestors|
       |$lisplibModemap| |$lisplibModemapAlist| |$lisplibSlot1|
       |$lisplibOperationAlist| |$lisplibSuperDomain| |$libFile|
       |$lisplibVariableAlist| |$lisplibCategory| op libname res ok filearg)
 (declare (special $lisplib |$op| |$lisplibAttributes| |$newConlist|
                   |$lisplibPredicates| |$lisplibCategoriesExtended|
                   |$lisplibForm| |$lisplibKind| |$algebraOutputStream|
                   |$lisplibAbbreviation| |$lisplibParents| |$spadLibFT|
                   |$lisplibAncestors| |$lisplibModemap| $filep
                   |$lisplibModemapAlist| |$lisplibSlot1|
```

```
                 |$lisplibOperationAlist| |$lisplibSuperDomain|
                 |$libFile| |$lisplibVariableAlist|
                 |$lisplibCategory| |$compileDocumentation|))
 (when (eq (car df) 'def) (car df))
 (setq op (caadr df))
 (|sayMSG| (|fillerSpaces| 72 "-"))
 (setq $lisplib t)
 (setq |$op| op)
 (setq |$lisplibAttributes| nil)
 (setq |$lisplibPredicates| nil)
 (setq |$lisplibCategoriesExtended| nil)
 (setq |$lisplibForm| nil)
 (setq |$lisplibKind| nil)
 (setq |$lisplibAbbreviation| nil)
 (setq |$lisplibParents| nil)
 (setq |$lisplibAncestors| nil)
 (setq |$lisplibModemap| nil)
 (setq |$lisplibModemapAlist| nil)
 (setq |$lisplibSlot1| nil)
 (setq |$lisplibOperationAlist| nil)
 (setq |$lisplibSuperDomain| nil)
 (setq |$libFile| nil)
 (setq |$lisplibVariableAlist| nil)
 (setq |$lisplibCategory| nil)
 (setq libname (|getConstructorAbbreviation| op))
 (cond
  ((and (boundp '|$compileDocumentation|) |$compileDocumentation|)
     (|compileDocumentation| libname))
  (t
   (|sayMSG| (cons "   initializing " (cons |$spadLibFT|
             (append (|bright| libname) (cons "for" (|bright| op))))))
   (|initializeLisplib| libname)
   (|sayMSG|
    (cons "   compiling into " (cons |$spadLibFT| (|bright| libname))))
   (setq ok nil)
   (unwind-protect
    (progn
     (setq res (funcall fn df m env prefix fal))
     (|sayMSG| (cons "   finalizing " (cons |$spadLibFT| (|bright| libname))))
     (|finalizeLisplib| libname)
     (setq ok t))
     (rshut |$libFile|))
   (when ok (|lisplibDoRename| libname))
   (setq filearg ($filep libname |$spadLibFT| 'a))
   (rpackfile filearg)
   (fresh-line |$algebraOutputStream|)
   (|sayMSG| (|fillerSpaces| 72 "-"))
   (|unloadOneConstructor| op libname)
   (localdatabase (list (getdatabase op 'abbreviation)) nil)
   (setq |$newConlist| (cons op |$newConlist|))
   (when (eq |$lisplibKind| '|category|)
    (|updateCategoryFrameForCategory| op)
    (|updateCategoryFrameForConstructor| op))
   res))))
```

———-

6.1.14 defun compileDocumentation

[makeInputFilename p??]
[rdefiostream p??]
[lisplibWrite p199]
[finalizeDocumentation p444]
[rshut p??]
[rpackfile p??]
[replaceFile p??]
[$fcopy p??]
[$spadLibFT p??]
[$EmptyMode p166]
[$e p??]

— defun compileDocumentation —

```
(defun |compileDocumentation| (libName)
 (let (filename stream)
 (declare (special |$e| |$EmptyMode| |$spadLibFT| $fcopy))
  (setq filename (makeInputFilename libName |$spadLibFT|))
  ($fcopy filename (cons libname (list 'doclb)))
  (setq stream
   (rdefiostream (cons (list 'file libName 'doclb) (list (cons 'mode 'o)))))
  (|lisplibWrite| "documentation" (|finalizeDocumentation|) stream)
  (rshut stream)
  (rpackfile (list libName 'doclb))
  (replaceFile (list libName |$spadLibFT|) (list libName 'doclb))
  (list '|dummy| |$EmptyMode| |$e|)))
```

———-

6.1.15 defun compArgumentConditions

[compOrCroak p528]
[$Boolean p??]
[$argumentConditionList p??]
[$argumentConditionList p??]

— defun compArgumentConditions —

```
(defun |compArgumentConditions| (env)
 (let (n a x y tmp1)
 (declare (special |$Boolean| |$argumentConditionList|))
  (setq |$argumentConditionList|
   (loop for item in |$argumentConditionList|
    do
     (setq n (first item))
     (setq a (second item))
```

```
     (setq x (third item))
     (setq y (subst a '|#1| x :test #'equal))
     (setq tmp1 (|compOrCroak| y |$Boolean| env))
     (setq env (third tmp1))
    collect
     (list n x (first tmp1))))
  env))
```

———-

6.1.16 defun compileCases

[eval p??]
[compile p163]
[getSpecialCaseAssoc p280]
[get p??]
[assocleft p??]
[outerProduct p??]
[assocright p??]
[mkpf p??]
[$getDomainCode p??]
[$insideFunctorIfTrue p??]
[$specialCaseKeyList p??]

— defun compileCases —

```
(defun |compileCases| (x |$e|)
 (declare (special |$e|))
 (labels (
  (isEltArgumentIn (Rlist x)
    (cond
     ((atom x) nil)
     ((and (consp x) (eq (qfirst x) 'elt) (consp (qrest x))
           (consp (qcddr x)) (eq (qcdddr x) nil))
      (or (member (second x) Rlist)
          (isEltArgumentIn Rlist (cdr x))))
     ((and (consp x) (eq (qfirst x) 'qrefelt) (consp (qrest x))
           (consp (qcddr x)) (eq (qcdddr x) nil))
      (or (member (second x) Rlist)
          (isEltArgumentIn Rlist (cdr x))))
     (t
      (or (isEltArgumentIn Rlist (car x))
          (isEltArgumentIn Rlist (CDR x))))))
  (FindNamesFor (r rp)
   (let (v u)
   (declare (special |$getDomainCode|))
    (cons r
     (loop for item in |$getDomainCode|
      do
        (setq v (second item))
        (setq u (third item))
      when (and (equal (second u) r) (|eval| (subst rp r u :test #'equal)))
```

```
     collect v)))))
 (let (|$specialCaseKeyList| specialCaseAssoc listOfDomains listOfAllCases cl)
 (declare (special |$specialCaseKeyList| |$true| |$insideFunctorIfTrue|))
  (setq |$specialCaseKeyList| nil)
  (cond
   ((null (eq |$insideFunctorIfTrue| t)) (|compile| x))
   (t
     (setq specialCaseAssoc
      (loop for y in (|getSpecialCaseAssoc|)
       when (and (null (|get| (first y) '|specialCase| |$e|))
                 (isEltArgumentIn (FindNamesFor (first y) (second y)) x))
       collect y))
      (cond
       ((null specialCaseAssoc) (|compile| x))
       (t
         (setq listOfDomains (assocleft specialCaseAssoc))
         (setq listOfAllCases (|outerProduct| (assocright specialCaseAssoc)))
         (setq cl
          (loop for z in listOfAllCases
           collect
            (progn
             (setq |$specialCaseKeyList|
              (loop for d in listOfDomains for c in z
               collect (cons d c)))
              (cons
               (mkpf
                (loop for d in listOfDomains for c in z
                 collect (list 'equal d c))
                'and)
                (list (|compile| (copy x)))))))
         (setq |$specialCaseKeyList| nil)
         (cons 'cond (append cl (list (list |$true| (|compile| x))))))))))
```

————-

6.1.17 defun compFunctorBody

[bootStrapError p204]
[compOrCroak p528]
[/editfile p??]
[$NRTaddForm p??]
[$functorForm p??]
[$bootStrapMode p??]

— defun compFunctorBody —

```
(defun |compFunctorBody| (form mode env parForm)
 (declare (ignore parForm))
 (let (tt)
 (declare (special |$NRTaddForm| |$functorForm| |$bootStrapMode| /editfile))
  (if |$bootStrapMode|
   (list (|bootStrapError| |$functorForm| /editfile) mode env)
```

```
(progn
 (setq tt (|compOrCroak| form mode env))
 (if (and (consp form)  (member (qfirst form) '(|add| capsule)))
  tt
  (progn
   (setq |$NRTaddForm|
    (if  (and (consp form) (eq (qfirst form) '|SubDomain|)
              (consp (qrest form)) (consp (qcddr form))
              (eq (qcdddr form) nil))
      (qsecond form)
      form))
   tt))))))
```

———-

6.1.18 defun compile

[member p??]
[getmode p??]
[get p??]
[modeEqual p335]
[userError p??]
[encodeItem p173]
[strconc p??]
[encodeFunctionName p172]
[splitEncodedFunctionName p173]
[sayBrightly p??]
[optimizeFunctionDef p212]
[putInLocalDomainReferences p177]
[constructMacro p174]
[spadCompileOrSetq p175]
[elapsedTime p??]
[addStats p??]
[printStats p??]
[$functionStats p??]
[$macroIfTrue p??]
[$doNotCompileJustPrint p??]
[$insideCapsuleFunctionIfTrue p??]
[$saveableItems p??]
[$lisplibItemsAlreadyThere p??]
[$splitUpItemsAlreadyThere p??]
[$lisplib p??]
[$compileOnlyCertainItems p??]
[$functorForm p??]
[$signatureOfForm p??]
[$suffix p??]
[$prefix p??]
[$signatureOfForm p??]
[$e p??]
[$functionStats p??]

[$savableItems p??]
[$suffix p??]

— defun compile —

```
(defun |compile| (u)
 (labels (
  (isLocalFunction (op)
   (let (tmp1)
   (declare (special |$e| |$formalArgList|))
    (and (null (|member| op |$formalArgList|))
         (progn
          (setq tmp1 (|getmode| op |$e|))
          (and (consp tmp1) (eq (qfirst tmp1) '|Mapping|)))))))
 (let (op lamExpr DC sig sel opexport opmodes opp parts s tt unew
       optimizedBody stuffToCompile result functionStats)
 (declare (special |$functionStats| |$macroIfTrue| |$doNotCompileJustPrint|
                   |$insideCapsuleFunctionIfTrue| |$saveableItems| |$e|
                   |$lisplibItemsAlreadyThere| |$splitUpItemsAlreadyThere|
                   |$compileOnlyCertainItems| $LISPLIB |$suffix|
                   |$signatureOfForm| |$functorForm| |$prefix|
                   |$savableItems|))
   (setq op (first u))
   (setq lamExpr (second u))
   (when |$suffix|
    (setq |$suffix| (1+ |$suffix|))
    (setq opp
     (progn
      (setq opexport nil)
      (setq opmodes
       (loop for item in (|get| op '|modemap| |$e|)
        do
         (setq dc (caar item))
         (setq sig (cdar item))
         (setq sel (cadadr item))
        when (and (eq dc '$)
                    (setq opexport t)
                    (let ((result t))
                     (loop for x in sig for y in |$signatureOfForm|
                      do (setq result (|modeEqual| x y)))
                     result))
        collect sel))
      (cond
       ((isLocalFunction op)
        (when opexport
         (|userError| (list op " is local and exported")))
        (intern (strconc (|encodeItem| |$prefix|) ";" (|encodeItem| op))))
       (t
        (|encodeFunctionName| op |$functorForm| |$signatureOfForm|
                              '|;| |$suffix|)))))
    (setq u (list opp lamExpr)))
   (when (and $lisplib |$compileOnlyCertainItems|)
    (setq parts (|splitEncodedFunctionName| (elt u 0) '|;|))
    (cond
```

```
     ((eq parts '|inner|)
       (setq |$savableItems| (cons (elt u 0) |$savableItems|)))
     (t
       (setq unew nil)
       (loop for item in |$splitUpItemsAlreadyThere|
        do
         (setq s (first item))
         (setq tt (second item))
         (when
          (and (equal (elt parts 0) (elt s 0))
               (equal (elt parts 1) (elt s 1))
               (equal (elt parts 2) (elt s 2)))
            (setq unew tt)))
       (cond
        ((null unew)
         (|sayBrightly| (list "   Error: Item did not previously exist"))
         (|sayBrightly| (cons "   Item not saved: " (|bright| (elt u 0))))
         (|sayBrightly|
           (list "   What's there is: " |$lisplibItemsAlreadyThere|))
         nil)
        (t
         (|sayBrightly| (list "   Renaming " (elt u 0) " as " unew))
         (setq u (cons unew (cdr u)))
         (setq |$savableItems| (cons unew |$saveableItems|)))))))
   (setq optimizedBody (|optimizeFunctionDef| u))
   (setq stuffToCompile
    (if |$insideCapsuleFunctionIfTrue|
     (|putInLocalDomainReferences| optimizedBody)
     optimizedBody))
   (cond
    ((eq |$doNotCompileJustPrint| t)
      (prettyprint stuffToCompile)
      opp)
    (|$macroIfTrue| (|constructMacro| stuffToCompile))
    (t
     (setq result (|spadCompileOrSetq| stuffToCompile))
     (setq functionStats (list 0 (|elapsedTime|)))
     (setq |$functionStats| (|addStats| |$functionStats| functionStats))
     (|printStats| functionStats)
      result)))))
```

———-

6.1.19 defvar $NoValueMode

— initvars —

```
(defvar |$NoValueMode| '|NoValueMode|)
```

———-

6.1.20 defvar $EmptyMode

`$EmptyMode` is a contant whose value is `$EmptyMode`. It is used by isPartialMode to decide if a modemap is partially constructed. If the `$EmptyMode` constant occurs anywhere in the modemap structure at any depth then the modemap is still incomplete. To find this constant the isPartialMode function calls CONTAINED `$EmptyMode` Y which will walk the structure Y looking for this constant.

— **initvars** —

```
(defvar |$EmptyMode| '|EmptyMode|)
```

———

6.1.21 defun hasFullSignature

TPDHERE: test with BASTYPE [get p**??**]

— **defun hasFullSignature** —

```
(defun |hasFullSignature| (argl signature env)
 (let (target ml u)
  (setq target (first signature))
  (setq ml (rest signature))
  (when target
   (setq u
     (loop for x in argl for m in ml
      collect (or m (|get| x '|mode| env) (return 'failed))))
   (unless (eq u 'failed) (cons target u)))))
```

———

6.1.22 defun addEmptyCapsuleIfNecessary

[$SpecialDomainNames p**??**]

— **defun addEmptyCapsuleIfNecessary** —

```
(defun |addEmptyCapsuleIfNecessary| (target rhs)
 (declare (special |$SpecialDomainNames|) (ignore target))
 (if (member (ifcar rhs) |$SpecialDomainNames|)
   rhs
   (list '|add| rhs (list 'capsule))))
```

———

6.1.23 defun getTargetFromRhs

[stackSemanticError p**??**]
[getTargetFromRhs p166]
[compOrCroak p528]

— **defun getTargetFromRhs** —

```
(defun |getTargetFromRhs| (lhs rhs env)
 (declare (special |$EmptyMode|))
  (cond
   ((and (consp rhs) (eq (qfirst rhs) 'capsule))
     (|stackSemanticError|
      (list "target category of " lhs
            " cannot be determined from definition")
     nil))
   ((and (consp rhs) (eq (qfirst rhs) '|SubDomain|) (consp (qrest rhs)))
    (|getTargetFromRhs| lhs (second rhs) env))
   ((and (consp rhs) (eq (qfirst rhs) '|add|)
         (consp (qrest rhs)) (consp (qcddr rhs))
         (eq (qcdddr rhs) nil)
         (consp (qthird rhs))
         (eq (qcaaddr rhs) 'capsule))
     (|getTargetFromRhs| lhs (second rhs) env))
   ((and (consp rhs) (eq (qfirst rhs) '|Record|))
     (cons '|RecordCategory| (rest rhs)))
   ((and (consp rhs) (eq (qfirst rhs) '|Union|))
     (cons '|UnionCategory| (rest rhs)))
   ((and (consp rhs) (eq (qfirst rhs) '|List|))
     (cons '|ListCategory| (rest rhs)))
   ((and (consp rhs) (eq (qfirst rhs) '|Vector|))
     (cons '|VectorCategory| (rest rhs)))
   (t
     (second (|compOrCroak| rhs |$EmptyMode| env)))))
```

———

6.1.24 defun giveFormalParametersValues

[put p??]
[get p??]

— **defun giveFormalParametersValues** —

```
(defun |giveFormalParametersValues| (argl env)
 (dolist (x argl)
  (setq env
   (|put| x '|value|
      (list (|genSomeVariable|) (|get| x '|mode| env) nil) env)))
 env)
```

———

6.1.25 defun macroExpandInPlace

[macroExpand p168]

— defun macroExpandInPlace —

```
(defun |macroExpandInPlace| (form env)
 (let (y)
  (setq y (|macroExpand| form env))
  (if (or (atom form) (atom y))
    y
    (progn
      (rplaca form (car y))
      (rplacd form (cdr y))
      form
    ))))
```

———-

6.1.26 defun macroExpand

[macroExpand p168]
[macroExpandList p168]

— defun macroExpand —

```
(defun |macroExpand| (form env)
 (let (u)
 (cond
  ((atom form)
   (if (setq u (|get| form '|macro| env))
    (|macroExpand| u env)
    form))
  ((and (consp form) (eq (qfirst form) 'def)
        (consp (qrest form))
        (consp (qcddr form))
        (consp (qcdddr form))
        (consp (qcddddr form))
        (eq (qrest (qcddddr form)) nil))
   (list 'def (|macroExpand| (second form) env)
              (|macroExpandList| (third form) env)
              (|macroExpandList| (fourth form) env)
              (|macroExpand| (fifth form) env)))
  (t (|macroExpandList| form env)))))
```

———-

6.1.27 defun macroExpandList

[macroExpand p168]
[getdatabase p??]

— defun macroExpandList —

```
(defun |macroExpandList| (lst env)
 (let (tmp)
```

```
  (if (and (consp lst) (eq (qrest lst) nil)
           (identp (qfirst lst)) (getdatabase (qfirst lst) 'niladic)
           (setq tmp (|get| (qfirst lst) '|macro| env)))
    (|macroExpand| tmp env)
    (loop for x in lst collect (|macroExpand| x env)))))
```

———-

6.1.28 defun makeCategoryPredicates

[$FormalMapVariableList p249]
[$TriangleVariableList p??]
[$mvl p??]
[$tvl p??]

— defun makeCategoryPredicates —

```
(defun |makeCategoryPredicates| (form u)
 (labels (
  (fn (u pl)
   (declare (special |$tvl| |$mvl|))
   (cond
    ((and (consp u) (eq (qfirst u) '|Join|) (consp (qrest u)))
      (fn (car (reverse (qrest u))) pl))
    ((and (consp u) (eq (qfirst u) '|has|))
      (|insert| (eqsubstlist |$mvl| |$tvl| u) pl))
    ((and (consp u) (member (qfirst u) '(signature attribute))) pl)
    ((atom u) pl)
    (t (fnl u pl))))
  (fnl (u pl)
   (dolist (x u) (setq pl (fn x pl)))
   pl))
 (declare (special |$FormalMapVariableList| |$mvl| |$tvl|
                   |$TriangleVariableList|))
  (setq |$tvl| (take (|#| (cdr form)) |$TriangleVariableList|))
  (setq |$mvl| (take (|#| (cdr form)) (cdr |$FormalMapVariableList|)))
  (fn u nil)))
```

———-

6.1.29 defun mkCategoryPackage

[strconc p??]
[pname p??]
[getdatabase p??]
[abbreviationsSpad2Cmd p??]
[JoinInner p??]
[assoc p??]
[sublislis p??]
[$options p??]
[$categoryPredicateList p??]

[$e p??]
[$FormalMapVariableList p249]

— defun mkCategoryPackage —

```
(defun |mkCategoryPackage| (form cat def)
 (labels (
  (fn (x oplist)
   (cond
    ((atom x) oplist)
    ((and (consp x) (eq (qfirst x) 'def) (consp (qrest x)))
      (cons (second x) oplist))
    (t
     (fn (cdr x) (fn (car x) oplist)))))
  (gn (cat)
   (cond
    ((and (consp cat) (eq (qfirst cat) 'category)) (cddr cat))
    ((and (consp cat) (eq (qfirst cat) '|Join|))   (gn (|last| (qrest cat))))
    (t nil))))
 (let (|$options| op argl packageName packageAbb nameForDollar packageArgl
      capsuleDefAlist explicitCatPart catvec fullCatOpList op1 sig
      catOpList packageCategory nils packageSig)
  (declare (special |$options| |$categoryPredicateList| |$e|
                    |$FormalMapVariableList|))
  (setq op (car form))
  (setq argl (cdr form))
  (setq packageName (intern (strconc (pname op) "&")))
  (setq packageAbb  (intern (strconc (getdatabase op 'abbreviation) "-")))
  (setq |$options| nil)
  (|abbreviationsSpad2Cmd| (list '|domain| packageAbb packageName))
  (setq nameForDollar (car (setdifference '(s a b c d e f g h i) argl)))
  (setq packageArgl (cons nameForDollar argl))
  (setq capsuleDefAlist (fn def nil))
  (setq explicitCatPart (gn cat))
  (setq catvec (|eval| (|mkEvalableCategoryForm| form)))
  (setq fullCatOpList (elt (|JoinInner| (list catvec) |$e|) 1))
  (setq catOpList
   (loop for x in fullCatOpList do
     (setq op1 (caar x))
     (setq sig (cadar x))
    when (|assoc| op1 capsuleDefAlist)
    collect (list 'signature op1 sig)))
  (when catOpList
   (setq packageCategory
    (cons 'category
     (cons '|domain| (sublislis argl |$FormalMapVariableList| catOpList))))
   (setq nils (loop for x in argl collect nil))
   (setq packageSig (cons packageCategory (cons form nils)))
   (setq |$categoryPredicateList|
     (subst nameForDollar '$ |$categoryPredicateList| :test #'equal))
   (subst nameForDollar '$
     (list 'def (cons packageName packageArgl)
           packageSig (cons nil nils) def)  :test #'equal)))))
```

———-

6.1.30 defun mkEvalableCategoryForm

[mkEvalableCategoryForm p171]
[compOrCroak p528]
[getdatabase p??]
[get p??]
[mkq p??]
[$Category p??]
[$e p??]
[$EmptyMode p166]
[$CategoryFrame p??]
[$Category p??]
[$CategoryNames p??]
[$e p??]

— defun mkEvalableCategoryForm —

```
(defun |mkEvalableCategoryForm| (c)
 (let (op argl tmp1 x m)
 (declare (special |$Category| |$e| |$EmptyMode| |$CategoryFrame|
                   |$CategoryNames|))
  (if (consp c)
   (progn
    (setq op (qfirst c))
    (setq argl (qrest c))
    (cond
     ((eq op '|Join|)
       (cons '|Join|
        (loop for x in argl
         collect (|mkEvalableCategoryForm| x))))
     ((eq op '|DomainSubstitutionMacro|)
       (|mkEvalableCategoryForm| (cadr argl)))
     ((eq op '|mkCategory|) c)
     ((member op |$CategoryNames|)
       (setq tmp1 (|compOrCroak| c |$EmptyMode| |$e|))
       (setq x (car tmp1))
       (setq m (cadr tmp1))
       (setq |$e| (caddr tmp1))
       (when (equal m |$Category|) x))
     ((or (eq (getdatabase op 'constructorkind) '|category|)
          (|get| op '|isCategory| |$CategoryFrame|))
       (cons op
        (loop for x in argl
         collect (mkq x))))
     (t
       (setq tmp1 (|compOrCroak| c |$EmptyMode| |$e|))
       (setq x (car tmp1))
       (setq m (cadr tmp1))
       (setq |$e| (caddr tmp1))
       (when (equal m |$Category|) x))))
```

```
   (mkq c))))
```

———-

6.1.31 defun encodeFunctionName

Code for encoding function names inside package or domain [mkRepititionAssoc p172]
[encodeItem p173]
[internl p??]
[getAbbreviation p277]
[length p??]
[$lisplib p??]
[$lisplibSignatureAlist p??]
[$lisplibSignatureAlist p??]

— defun encodeFunctionName —

```
(defun |encodeFunctionName| (fun package signature sep count)
 (let (packageName arglist signaturep reducedSig n x encodedSig encodedName)
 (declare (special |$lisplibSignatureAlist| $lisplib))
  (setq packageName (car package))
  (setq arglist (cdr package))
  (setq signaturep (subst '$ package signature  :test #'equal))
  (setq reducedSig
   (|mkRepititionAssoc| (append (cdr signaturep) (list (car signaturep)))))
  (setq encodedSig
   (let ((result ""))
    (loop for item in reducedSig
     do
      (setq n (car item))
      (setq x (cdr item))
      (setq result
       (strconc result
        (if (eql n 1)
          (|encodeItem| x)
          (strconc (princ-to-string n) (|encodeItem| x))))))
     result))
  (setq encodedName
   (internl (|getAbbreviation| packageName (|#| arglist))
            '|;| (|encodeItem| fun) '|;| encodedSig sep (princ-to-string count)))
  (when $lisplib
   (setq |$lisplibSignatureAlist|
     (cons (cons encodedName signaturep) |$lisplibSignatureAlist|)))
  encodedName))
```

———-

6.1.32 defun mkRepititionAssoc

[mkRepfun p??]

— defun mkRepititionAssoc —

```
(defun |mkRepititionAssoc| (z)
 (labels (
  (mkRepfun (z n)
    (cond
     ((null z) nil)
     ((and (consp z) (eq (qrest z) nil) (list (cons n (qfirst z)))))
     ((and (consp z) (consp (qrest z)) (equal (qsecond z) (qfirst z)))
      (mkRepfun (cdr z) (1+ n)))
     (t (cons (cons n (car z)) (mkRepfun (cdr z) 1))))))
 (mkRepfun z 1)))
```

———

6.1.33 defun splitEncodedFunctionName

[strpos p??]

— defun splitEncodedFunctionName —

```
(defun |splitEncodedFunctionName| (encodedName sep)
 (let (sep0 p1 p2 p3 s1 s2 s3 s4)
  ; sep0 is the separator used in "encodeFunctionName".
  (setq sep0 ";")
  (unless (stringp encodedName) (setq encodedName (princ-to-string encodedName)))
  (cond
   ((null (setq p1 (strpos sep0 encodedName 0 "*"))) nil)
   ; This is picked up in compile for inner functions in partial compilation
   ((null (setq p2 (strpos sep0 encodedName (1+ p1) "*"))) '|inner|)
   ((null (setq p3 (strpos sep encodedName (1+ p2) "*"))) nil)
   (t
    (setq s1 (substring encodedName 0 p1))
    (setq s2 (substring encodedName (1+ p1) (- p2 p1 1)))
    (setq s3 (substring encodedName (1+ p2) (- p3 p2 1)))
    (setq s4 (substring encodedName (1+ p3) nil))
    (list s1 s2 s3 s4)))))
```

———

6.1.34 defun encodeItem

[getCaps p174]
[identp p??]
[pname p??]

— defun encodeItem —

```
(defun |encodeItem| (x)
 (cond
  ((consp x) (|getCaps| (qfirst x)))
  ((identp x) (pname x))
```

```
  (t (princ-to-string x))))
```

———-

6.1.35 defun getCaps

[maxindex p??]
[downcase p??]
[strconc p??]

— **defun getCaps** —

```
(defun |getCaps| (x)
 (let (s c clist tmp1)
  (setq s (princ-to-string x))
  (setq clist
   (loop for i from 0 to (maxindex s)
    when (upper-case-p (setq c (elt s i)))
    collect c))
  (cond
   ((null clist) "_")
   (t
    (setq tmp1
     (cons (first clist) (loop for u in (rest clist) collect (downcase u))))
    (let ((result ""))
     (loop for u in tmp1
      do (setq result (strconc result u)))
     result)))))
```

———-

6.1.36 defun constructMacro

constructMacro (form is [nam,[lam,vl,body]]) [stackSemanticError p??]
[identp p??]

— **defun constructMacro** —

```
(defun |constructMacro| (form)
 (let (vl body)
  (setq vl (cadadr form))
  (setq body (car (cddadr form)))
  (cond
   ((null (let ((result t))
           (loop for x in vl
            do (setq result (and result (atom x))))
           result))
     (|stackSemanticError| (list '|illegal parameters for macro: | vl) nil))
   (t
     (list 'xlam (loop for x in vl when (identp x) collect x) body)))))
```

———-

6.1.37 defun spadCompileOrSetq

[contained p??]
[sayBrightly p??]
[bright p??]
[LAM,EVALANDFILEACTQ p??]
[mkq p??]
[comp p530]
[compileConstructor p176]
[$insideCapsuleFunctionIfTrue p??]

— defun spadCompileOrSetq —

```
(defun |spadCompileOrSetq| (form)
 (let (nam lam vl body namp tmp1 e vlp macform)
 (declare (special |$insideCapsuleFunctionIfTrue|))
  (setq nam (car form))
  (setq lam (caadr form))
  (setq vl (cadadr form))
  (setq body (car (cddadr form)))
  (cond
   ((and (consp vl) (progn (setq tmp1 (reverse vl)) t)
         (consp tmp1)
         (progn
          (setq e (qfirst tmp1))
          (setq vlp (qrest tmp1))
          t)
         (progn (setq vlp (nreverse vlp)) t)
         (consp body)
         (progn (setq namp (qfirst body)) t)
         (equal (qrest body) vlp))
     (|LAM,EVALANDFILEACTQ|
      (list 'put (mkq nam) (mkq '|SPADreplace|) (mkq namp)))
     (|sayBrightly|
      (cons "     " (append (|bright| nam)
       (cons "is replaced by" (|bright| namp))))))
   ((and (or (atom body)
             (let ((result t))
              (loop for x in body
               do (setq result (and result (atom x))))
              result))
         (consp vl)
         (progn (setq tmp1 (reverse vl)) t)
         (consp tmp1)
         (progn
          (setq e (qfirst tmp1))
          (setq vlp (qrest tmp1))
          t)
         (progn (setq vlp (nreverse vlp)) t)
         (null (contained e body)))
    (setq macform (list 'xlam vlp body))
```

```
    (|LAM,EVALANDFILEACTQ|
     (list 'put (mkq nam) (mkq '|SPADreplace|) (mkq macform)))
    (|sayBrightly| (cons "     " (append (|bright| nam)
      (cons "is replaced by" (|bright| body))))))
   (t nil))
  (if |$insideCapsuleFunctionIfTrue|
   (car (comp (list form)))
   (|compileConstructor| form))))
```

———–

6.1.38 defun compileConstructor

[compileConstructor1 p176]
[clearClams p??]

— defun compileConstructor —

```
(defun |compileConstructor| (form)
 (let (u)
  (setq u (|compileConstructor1| form))
  (|clearClams|)
  u))
```

———–

6.1.39 defun compileConstructor1

[getdatabase p??]
[compAndDefine p177]
[comp p530]
[clearConstructorCache p??]
[$mutableDomain p??]
[$ConstructorCache p??]
[$clamList p??]
[$clamList p??]

— defun compileConstructor1 —

```
(defun |compileConstructor1| (form)
 (let (|$clamList| fn key vl bodyl lambdaOrSlam compForm u)
 (declare (special |$clamList| |$ConstructorCache| |$mutableDomain|))
  (setq fn (car form))
  (setq key (caadr form))
  (setq vl (cadadr form))
  (setq bodyl (cddadr form))
  (setq |$clamList| nil)
  (setq lambdaOrSlam
   (cond
    ((eq (getdatabase fn 'constructorkind) '|category|) 'spadslam)
    (|$mutableDomain| 'lambda)
```

```
    (t
     (setq |$clamList|
      (cons (list fn '|$ConstructorCache| '|domainEqualList| '|count|)
            |$clamList|))
     'lambda)))
  (setq compForm (list (list fn (cons lambdaorslam (cons vl bodyl)))))
  (if (eq (getdatabase fn 'constructorkind) '|category|)
   (setq u (|compAndDefine| compForm))
   (setq u (comp compForm)))
  (|clearConstructorCache| fn)
  (car u)))
```

———-

6.1.40 defun compAndDefine

This function is used but never defined. We define a dummy function here. All references to it should be removed. **TPDHERE: This function is used but never defined. Remove it.**

— defun compAndDefine —

```
(defun compAndDefine (arg)
 (declare (ignore arg))
 nil)
```

———-

6.1.41 defun putInLocalDomainReferences

[NRTputInTail p178]
[$QuickCode p**??**]
[$elt p285]

— defun putInLocalDomainReferences —

```
(defun |putInLocalDomainReferences| (def)
 (let (|$elt| opName lam varl body)
 (declare (special |$elt| |$QuickCode|))
  (setq opName (car def))
  (setq lam (caadr def))
  (setq varl (cadadr def))
  (setq body (car (cddadr def)))
  (setq |$elt| (if |$QuickCode| 'qrefelt 'elt))
  (|NRTputInTail| (cddadr def))
  def))
```

———-

6.1.42 defun NRTputInTail

[lassoc p??]
[NRTassocIndex p317]
[rplaca p??]
[NRTputInHead p178]
[$elt p285]
[$devaluateList p??]

— defun NRTputInTail —

```
(defun |NRTputInTail| (x)
 (let (u k)
 (declare (special |$elt| |$devaluateList|))
  (maplist #'(lambda (y)
              (cond
               ((atom (setq u (car y)))
                 (cond
                  ((or (eq u '$) (lassoc u |$devaluateList|))
                    nil)
                  ((setq k (|NRTassocIndex| u))
                   (cond
                    ; u atomic means that the slot will always contain a vector
                    ((atom u) (rplaca y (list |$elt| '$ k)))
                    ; this reference must check that slot is a vector
                    (t (rplaca y (list 'spadcheckelt '$ k)))))
                  (t nil)))
               (t (|NRTputInHead| u))))
    x)
  x))
```

——————

6.1.43 defun NRTputInHead

[NRTputInTail p178]
[NRTassocIndex p317]
[NRTputInHead p178]
[lastnode p??]
[keyedSystemError p??]
[$elt p285]

— defun NRTputInHead —

```
(defun |NRTputInHead| (bod)
 (let (fn clauses dom tmp2 ind k)
 (declare (special |$elt|))
  (cond
   ((atom bod) bod)
   ((and (consp bod) (eq (qcar bod) 'spadcall) (consp (qcdr bod))
         (progn (setq tmp2 (reverse (qcdr bod))) t) (consp tmp2))
      (setq fn (qcar tmp2))
```

```
     (|NRTputInTail| (cdr bod))
     (cond
       ((and (consp fn) (consp (qcdr fn)) (consp (qcdr (qcdr fn)))
             (eq (qcdddr fn) nil) (null (eq (qsecond fn) '$))
            (member (qcar fn) '(elt qrefelt const)))
          (when (setq k (|NRTassocIndex| (qsecond fn)))
             (rplaca (lastnode bod) (list |$elt| '$ k))))
       (t (|NRTputInHead| fn) bod)))
  ((and (consp bod) (eq (qcar bod) 'cond))
     (setq clauses (qcdr bod))
     (loop for cc in clauses do (|NRTputInTail| cc))
     bod)
  ((and (consp bod) (eq (qcar bod) 'quote)) bod)
  ((and (consp bod) (eq (qcar bod) 'closedfn)) bod)
  ((and (consp bod) (eq (qcar bod) 'spadconst) (consp (qcdr bod))
        (consp (qcddr bod)) (eq (qcdddr bod) nil))
     (setq dom (qsecond bod))
     (setq ind (qthird bod))
     (rplaca bod |$elt|)
     (cond
       ((eq dom '$) nil)
       ((setq k (|NRTassocIndex| dom))
         (rplaca (lastnode bod) (list |$elt| '$ k))
         bod)
       (t
        (|keyedSystemError|
           "Unexpected error or improper call to system function %1: %2"
          (list "NRTputInHead" "unexpected SPADCONST form")))))
  (t
    (|NRTputInHead| (car bod))
    (|NRTputInTail| (cdr bod)) bod))))
```

———-

6.1.44 defun getArgumentModeOrMoan

[getArgumentMode p285]
[stackSemanticError p??]

— defun getArgumentModeOrMoan —

```
(defun |getArgumentModeOrMoan| (x form env)
 (or (|getArgumentMode| x env)
     (|stackSemanticError|
        (list '|argument | x '| of | form '| is not declared|) nil)))
```

———-

6.1.45 defun augLisplibModemapsFromCategory

[sublis p??]
[mkAlistOfExplicitCategoryOps p181]
[isCategoryForm p??]
[lassoc p??]
[member p??]
[mkpf p??]
[interactiveModemapForm p183]
[$lisplibModemapAlist p??]
[$EmptyEnvironment p??]
[$domainShell p??]
[$PatternVariableList p??]
[$lisplibModemapAlist p??]

— defun augLisplibModemapsFromCategory —

```
(defun |augLisplibModemapsFromCategory| (form body signature)
 (let (argl sl opAlist nonCategorySigAlist domainList catPredList op sig
       pred sel predp modemap)
 (declare (special |$lisplibModemapAlist| |$EmptyEnvironment|
                   |$domainShell| |$PatternVariableList|))
  (setq op (car form))
  (setq argl (cdr form))
  (setq sl
   (cons (cons '$ '*1)
    (loop for a in argl for p in (rest |$PatternVariableList|)
     collect (cons a p))))
  (setq form (sublis sl form))
  (setq body (sublis sl body))
  (setq signature (sublis sl signature))
  (when (setq opAlist (sublis sl (elt |$domainShell| 1)))
   (setq nonCategorySigAlist
    (|mkAlistOfExplicitCategoryOps| (subst '*1 '$ body :test #'equal)))
   (setq domainList
    (loop for a in (rest form) for m in (rest signature)
     when (|isCategoryForm| m |$EmptyEnvironment|)
     collect (list a m)))
  (setq catPredList
   (loop for u in (cons (list '*1 form) domainList)
    collect (cons '|ofCategory| u)))
  (loop for entry in opAlist
   when (|member| (cadar entry) (lassoc (caar entry) nonCategorySigAlist))
   do
    (setq op (caar entry))
    (setq sig (cadar entry))
    (setq pred (cadr entry))
    (setq sel (caddr entry))
    (setq predp (mkpf (cons pred catPredList) 'and))
    (setq modemap (list (cons '*1 sig) (list predp sel)))
    (setq |$lisplibModemapAlist|
      (cons (cons op (|interactiveModemapForm| modemap))
            |$lisplibModemapAlist|))))))
```

———-

6.1.46 defun mkAlistOfExplicitCategoryOps

[keyedSystemError p??]
[union p??]
[mkAlistOfExplicitCategoryOps p181]
[flattenSignatureList p182]
[nreverse0 p??]
[remdup p??]
[assocleft p??]
[isCategoryForm p??]
[$e p??]

— defun mkAlistOfExplicitCategoryOps —

```
(defun |mkAlistOfExplicitCategoryOps| (target)
 (labels (
  (atomizeOp (op)
   (cond
    ((atom op) op)
    ((and (consp op) (eq (qrest op) nil)) (qfirst op))
    (t (|keyedSystemError|
          "Unexpected error or improper call to system function %1: %2"
        (list "mkAlistOfExplicitCategoryOps" "bad signature")))))
  (fn (op u)
   (if (and (consp u) (consp (qfirst u)))
    (if (equal (qcaar u) op)
     (cons (qcdar u) (fn op (qrest u)))
     (fn op (qrest u))))))
 (let (z tmp1 op sig u opList)
 (declare (special |$e|))
  (when (and (consp target) (eq (qfirst target) '|add|) (consp (qrest target)))
    (setq target (second target)))
  (cond
   ((and (consp target) (eq (qfirst target) '|Join|))
    (setq z (qrest target))
    (PROG (tmp1)
     (RETURN
       (DO ((G167566 z (CDR G167566)) (cat nil))
           ((OR (ATOM G167566) (PROGN (setq cat (CAR G167566)) nil))
             tmp1)
         (setq tmp1 (|union| tmp1 (|mkAlistOfExplicitCategoryOps| cat)))))))
   ((and (consp target) (eq (qfirst target) 'category)
         (progn
           (setq tmp1 (qrest target))
           (and (consp tmp1)
                (progn (setq z (qrest tmp1)) t))))
     (setq z (|flattenSignatureList| (cons 'progn z)))
     (setq u
      (prog (G167577)
```

```
      (return
       (do ((G167583 z (cdr G167583)) (x nil))
           ((or (atom G167583)) (nreverse0 G167577))
         (setq x (car G167583))
         (cond
           ((and (consp x) (eq (qfirst x) 'signature) (consp (qrest x))
                  (consp (qcddr x)))
             (setq op (qsecond x))
             (setq sig (qthird x))
             (setq G167577 (cons (cons (atomizeOp op) sig) G167577))))))))
    (setq opList (remdup (assocleft u)))
    (prog (G167593)
     (return
      (do ((G167598 opList (cdr G167598)) (x nil))
          ((or (atom G167598)) (nreverse0 G167593))
         (setq x (car G167598))
         (setq G167593 (cons (cons x (fn x u)) G167593))))))
   ((|isCategoryForm| target |$e|) nil)
   (t
    (|keyedSystemError|
       "Unexpected error or improper call to system function %1: %2"
     (list "mkAlistOfExplicitCategoryOps" "bad signature")))))))
```

———-

6.1.47 defun flattenSignatureList

[flattenSignatureList p182]

— defun flattenSignatureList —

```
(defun |flattenSignatureList| (x)
 (let (zz)
  (cond
   ((atom x) nil)
   ((and (consp x) (eq (qfirst x) 'signature)) (list x))
   ((and (consp x) (eq (qfirst x) 'if) (consp (qrest x))
         (consp (qcddr x)) (consp (qcdddr x))
         (eq (qcddddr x) nil))
    (append (|flattenSignatureList| (third x))
            (|flattenSignatureList| (fourth x))))
   ((and (consp x) (eq (qfirst x) 'progn))
     (loop for x in (qrest x)
      do
        (if (and (consp x) (eq (qfirst x) 'signature))
          (setq zz (cons x zz))
          (setq zz (append (|flattenSignatureList| x) zz))))
     zz)
   (t nil))))
```

———-

6.1.48 defun interactiveModemapForm

Create modemap form for use by the interpreter. This function replaces all specific domains mentioned in the modemap with pattern variables, and predicates [replaceVars p184]
[modemapPattern p190]
[substVars p189]
[fixUpPredicate p184]
[$PatternVariableList p??]
[$FormalMapVariableList p249]

— defun interactiveModemapForm —

```
(defun |interactiveModemapForm| (mm)
 (labels (
  (fn (x)
    (if (and (consp x) (consp (qrest x))
             (consp (qcddr x)) (eq (qcdddr x) nil)
             (not (eq (qfirst x) '|isFreeFunction|))
             (atom (qthird x)))
     (list (first x) (second x) (list (third x)))
     x)))
 (let (pattern dc sig mmpat patternAlist partial patvars
       domainPredicateList tmp1 pred dependList cond)
 (declare (special |$PatternVariableList| |$FormalMapVariableList|))
  (setq mm
   (|replaceVars| (copy mm) |$PatternVariableList| |$FormalMapVariableList|))
  (setq pattern (car mm))
  (setq dc (caar mm))
  (setq sig (cdar mm))
  (setq pred (cadr mm))
  (setq pred
   (prog ()
    (return
     (do ((x pred (cdr x)) (result nil))
         ((atom x) (nreverse0 result))
      (setq result (cons (fn (car x)) result))))))
  (setq tmp1 (|modemapPattern| pattern sig))
  (setq mmpat (car tmp1))
  (setq patternAlist (cadr tmp1))
  (setq partial (caddr tmp1))
  (setq patvars (cadddr tmp1))
  (setq tmp1 (|substVars| pred patternAlist patvars))
  (setq pred (car tmp1))
  (setq domainPredicateList (cadr tmp1))
  (setq tmp1 (|fixUpPredicate| pred domainPredicateList partial (cdr mmpat)))
  (setq pred (car tmp1))
  (setq dependList (cdr tmp1))
  (setq cond (car pred))
  (list mmpat cond))))
```

———-

6.1.49 defun replaceVars

Replace every identifier in oldvars with the corresponding identifier in newvars in the expression x

— defun replaceVars —

```
(defun |replaceVars| (x oldvars newvars)
 (loop for old in oldvars for new in newvars
  do (setq x (subst new old x :test #'equal)))
 x)
```

———-

6.1.50 defun fixUpPredicate

[length p??]
[orderPredicateItems p185]
[moveORsOutside p188]

— defun fixUpPredicate —

```
(defun |fixUpPredicate| (predClause domainPreds partial sig)
 (let (predicate fn skip predicates tmp1 dependList pred)
  (setq predicate (car predClause))
  (setq fn (cadr predClause))
  (setq skip (cddr predClause))
  (cond
   ((eq (car predicate) 'and)
     (setq predicates (append domainPreds (cdr predicate))))
   ((not (equal predicate (mkq t)))
     (setq predicates (cons predicate domainPreds)))
   (t
     (setq predicates (or domainPreds (list predicate)))))
  (cond
   ((> (|#| predicates) 1)
     (setq pred (cons 'and predicates))
     (setq tmp1 (|orderPredicateItems| pred sig skip))
     (setq pred (car tmp1))
     (setq dependlist (cdr tmp1))
     tmp1)
   (t
     (setq pred (|orderPredicateItems| (car predicates) sig skip))
     (setq dependList
      (when (and (consp pred) (eq (qfirst pred) '|isDomain|)
                 (consp (qrest pred)) (consp (qcddr pred))
                 (eq (qcdddr pred) nil)
                 (consp (qthird pred))
                 (eq (qcdaddr pred) nil))
       (list (second pred))))))
  (setq pred (|moveORsOutside| pred))
  (when partial (setq pred (cons '|partial| pred)))
  (cons (cons pred (cons fn skip)) dependList)))
```

———-

6.1.51 defun orderPredicateItems

[signatureTran p185]
[orderPredTran p185]

— defun orderPredicateItems —

```
(defun |orderPredicateItems| (pred1 sig skip)
 (let (pred)
  (setq pred (|signatureTran| pred1))
  (if (and (consp pred) (eq (qfirst pred) 'and))
     (|orderPredTran| (qrest pred) sig skip)
     pred)))
```

———-

6.1.52 defun signatureTran

[signatureTran p185]
[isCategoryForm p??]
[$e p??]

— defun signatureTran —

```
(defun |signatureTran| (pred)
 (declare (special |$e|))
  (cond
   ((atom pred) pred)
   ((and (consp pred) (eq (qfirst pred) '|has|) (CONSP (qrest pred))
         (consp (qcddr pred))
         (eq (qcdddr pred) nil)
         (|isCategoryForm| (third pred) |$e|))
     (list '|ofCategory| (second pred) (third pred)))
   (t
    (loop for p in pred
     collect (|signatureTran| p)))))
```

———-

6.1.53 defun orderPredTran

[member p??]
[delete p??]
[unionq p??]
[listOfPatternIds p??]
[intersectionq p??]
[setdifference p??]
[insertWOC p??]

[isDomainSubst p188]

— defun orderPredTran —

```
(defun |orderPredTran| (oldList sig skip)
 (let (lastDependList somethingDone lastPreds indepvl depvl dependList
       noldList x ids fullDependList newList answer)
;  --(1) make two kinds of predicates appear last:
;  -----  (op *target ..) when *target does not appear later in sig
;  -----  (isDomain *1 ..)
  (SEQ
   (loop for pred in oldList
    do (cond
        ((or (and (consp pred) (consp (qrest pred))
                  (consp (qcddr pred))
                  (eq (qcdddr pred) nil)
                  (member (qfirst pred) '(|isDomain| |ofCategory|))
                  (equal (qsecond pred) (car sig))
                  (null (|member| (qsecond pred) (cdr sig))))
             (and (null skip) (consp pred) (eq (qfirst pred) '|isDomain|)
                  (consp (qrest pred)) (consp (qcddr pred))
                  (eq (qcdddr pred) nil)
                  (equal (qsecond pred) '*1)))
          (setq oldList (|delete| pred oldList))
          (setq lastPreds (cons pred lastPreds)))))
;  --(2a) lastDependList=list of all variables that lastPred forms depend upon
   (setq lastDependList
    (let (result)
     (loop for x in lastPreds
      do (setq result (unionq result (|listOfPatternIds| x))))
    result))
;  --(2b) dependList=list of all variables that isDom/ofCat forms depend upon
   (setq dependList
    (let (result)
     (loop for x in oldList
      do (when
          (and (consp x)
               (or (eq (qfirst x) '|isDomain|) (eq (qfirst x) '|ofCategory|))
               (consp (qrest x)) (consp (qcddr x))
               (eq (qcdddr x) nil))
           (setq result (unionq result (|listOfPatternIds| (third x))))))
     result))
;  --(3a) newList= list of ofCat/isDom entries that don't depend on
   (loop for x in oldList
    do
      (cond
       ((and (consp x)
             (or (eq (qfirst x) '|ofCategory|) (eq (qfirst x) '|isDomain|))
             (consp (qrest x)) (consp (qcddr x))
             (eq (qcdddr x) nil))
        (setq indepvl (|listOfPatternIds| (second x)))
        (setq depvl (|listOfPatternIds| (third x))))
       (t
         (setq indepvl (|listOfPatternIds| x))
```

```
        (setq depvl nil)))
      (when
       (and (null (intersectionq indepvl dependList))
            (intersectionq indepvl lastDependList))
          (setq somethingDone t)
          (setq lastPreds (append lastPreds (list x)))
          (setq oldList (|delete| x oldList))))
;  --(3b) newList= list of ofCat/isDom entries that don't depend on
   (loop while oldList do
    (loop for x in oldList do
     (cond
      ((and (consp x)
            (or (eq (qfirst x) '|ofCategory|) (eq (qfirst x) '|isDomain|))
            (consp (qrest x))
            (consp (qcddr x)) (eq (qcdddr x) nil))
       (setq indepvl (|listOfPatternIds| (second x)))
       (setq depvl (|listOfPatternIds| (third x))))
      (t
        (setq indepvl (|listOfPatternIds| x))
        (setq depvl nil)))
     (when (null (intersectionq indepvl dependList))
        (setq dependList (SETDIFFERENCE dependList depvl))
        (setq newList (APPEND newList (list x)))))
;  --(4) noldList= what is left over
    (cond
     ((equal (setq noldList (setdifference oldList newList)) oldList)
       (setq newList (APPEND newList oldList))
       (return nil))
     (t
       (setq oldList noldList))))
   (loop for pred in newList do
     (when
       (and (consp pred)
             (or (eq (qfirst pred) '|isDomain|) (eq (qfirst x) '|ofCategory|))
             (consp (qrest pred))
             (consp (qcddr pred))
             (eq (qcdddr pred) nil))
         (setq ids (|listOfPatternIds| (third pred)))
         (when
           (let (result)
             (loop for id in ids do
              (setq result (and result (|member| id fullDependList))))
             result)
           (setq fullDependList (|insertWOC| (second pred) fullDependList)))
         (setq fullDependList (unionq fullDependList ids))))
   (setq newList (append newList lastPreds))
   (setq newList (|isDomainSubst| newList))
   (setq answer
    (cons (cons 'and newList) (intersectionq fullDependList sig))))))
```

———-

6.1.54 defun isDomainSubst

— defun isDomainSubst —

```
(defun |isDomainSubst| (u)
 (labels (
  (findSub (x alist)
  (cond
   ((null alist) nil)
   ((and (consp alist) (consp (qfirst alist))
         (eq (qcaar alist) '|isDomain|)
         (consp (qcdar alist))
         (consp (qcddar alist))
         (eq (qcdddar alist) nil)
         (equal x (cadar alist)))
         (caddar alist))
    (t (findSub x (cdr alist)))))
  (fn (x alist)
   (let (s)
    (declare (special |$PatternVariableList|))
    (if (atom x)
     (if
      (and (identp x)
           (member x |$PatternVariableList|)
           (setq s (findSub x alist)))
         s
         x)
     (cons (car x)
      (loop for y in (cdr x)
       collect (fn y alist)))))))
 (let (head tail nhead)
  (if (consp u)
   (progn
    (setq head (qfirst u))
    (setq tail (qrest u))
    (setq nhead
     (cond
      ((and (consp head) (eq (qfirst head) '|isDomain|)
            (consp (qrest head)) (consp (qcddr head))
            (eq (qcdddr head) nil))
        (list '|isDomain| (second head)
           (fn (third head) tail)))
      (t head)))
     (cons nhead (|isDomainSubst| (cdr u))))
   u))))
```

———

6.1.55 defun moveORsOutside

[moveORsOutside p188]

— defun moveORsOutside —

```
(defun |moveORsOutside| (p)
 (let (q x)
  (cond
   ((and (consp p) (eq (qfirst p) 'and))
    (setq q
     (prog (G167169)
       (return
        (do ((G167174 (cdr p) (cdr G167174)) (|r| nil))
            ((or (atom G167174)) (nreverse0 G167169))
           (setq |r| (CAR G167174))
           (setq G167169 (cons (|moveORsOutside| |r|) G167169))))))
    (cond
     ((setq x
       (let (tmp1)
        (loop for r in q
         when (and (consp r) (eq (qfirst r) 'or))
         do (setq tmp1 (or tmp1 r)))
        tmp1))
       (|moveORsOutside|
        (cons 'or
         (let (tmp1)
          (loop for tt in (cdr x)
           do (setq tmp1 (cons (cons 'and (subst tt x q :test #'equal)) tmp1)))
          (nreverse0 tmp1)))))
     (t (cons 'and q))))
   (t p))))

;(defun |moveORsOutside| (p)
; (let (q s x tmp1)
; (cond
;  ((and (consp p) (eq (qfirst p) 'and))
;    (setq q (loop for r in (qrest p) collect (|moveORsOutside| r)))
;    (setq tmp1
;     (loop for r in q
;      when (and (consp r) (eq (qrest r) 'or))
;      collect r))
;    (setq x (mapcar #'(lambda (a b) (or a b)) tmp1))
;    (if x
;      (|moveORsOutside|
;       (cons 'or
;        (loop for tt in (cdr x)
;         collect (cons 'and (subst tt x q :test #'equal)))))
;      (cons 'and q)))
;   ('t p))))
```

———

6.1.56 defun substVars

Make pattern variable substitutions. [nsubst p??]
[contained p??]

[$FormalMapVariableList p249]

— defun substVars —

```
(defun |substVars| (pred patternAlist patternVarList)
 (let (patVar value everything replacementVar domainPredicates)
 (declare (special |$FormalMapVariableList|))
  (setq domainPredicates NIL)
  (maplist
   #'(lambda (x)
      (setq patVar (caar x))
      (setq value (cdar x))
      (setq pred (subst patVar value pred :test #'equal))
      (setq patternAlist (|nsubst| patVar value patternAlist))
      (setq domainPredicates
        (subst patVar value domainPredicates :test #'equal))
      (unless (member value |$FormalMapVariableList|)
       (setq domainPredicates
         (cons (list '|isDomain| patVar value) domainPredicates))))
     patternAlist)
  (setq everything (list pred patternAlist domainPredicates))
  (dolist (var |$FormalMapVariableList|)
    (cond
     ((contained var everything)
        (setq replacementVar (car patternVarList))
        (setq patternVarList (cdr patternVarList))
        (setq pred (subst replacementVar var pred :test #'equal))
        (setq domainPredicates
          (subst replacementVar var domainPredicates :test #'equal)))))
  (list pred domainPredicates)))
```

———–

6.1.57 defun modemapPattern

[rassoc p??]
[$PatternVariableList p??]

— defun modemapPattern —

```
(defun |modemapPattern| (mmPattern sig)
 (let (partial patvar patvars mmpat patternAlist)
 (declare (special |$PatternVariableList|))
   (setq patternAlist nil)
   (setq mmpat nil)
   (setq patvars |$PatternVariableList|)
   (setq partial nil)
   (maplist
    #'(lambda (xTails)
      (let ((x (car xTails)))
       (when  (and (consp x) (eq (qfirst x) '|Union|)
                  (consp (qrest x)) (consp (qcddr x))
                  (eq (qcdddr x) nil)
```

```
                  (equal (third x) "failed")
                  (equal xTails sig))
         (setq x (second x))
         (setq partial t))
       (setq patvar (|rassoc| x patternAlist))
       (cond
        ((null (null patvar))
         (setq mmpat (cons patvar mmpat)))
        (t
         (setq patvar (car patvars))
         (setq patvars (cdr patvars))
         (setq mmpat (cons patvar mmpat))
         (setq patternAlist (cons (cons patvar x) patternAlist))))))
     mmPattern)
   (list (nreverse mmpat) patternAlist partial patvars)))
```

———-

6.1.58 defun evalAndRwriteLispForm

[eval p??]
[rwriteLispForm p191]

— defun evalAndRwriteLispForm —

```
(defun |evalAndRwriteLispForm| (key form)
 (|eval| form)
 (|rwriteLispForm| key form))
```

———-

6.1.59 defun rwriteLispForm

[$libFile p??]
[$lisplib p??]

— defun rwriteLispForm —

```
(defun |rwriteLispForm| (key form)
 (declare (special |$libFile| $lisplib))
 (when $lisplib
   (|rwrite| key form |$libFile|)
   (|LAM,FILEACTQ| key form)))
```

———-

6.1.60 defun mkConstructor

[mkConstructor p191]

— defun mkConstructor —

```
(defun |mkConstructor| (form)
 (cond
  ((atom form) (list '|devaluate| form))
  ((null (rest form)) (list 'quote (list (first form))))
  (t
   (cons 'list
    (cons (mkq (first form))
     (loop for x in (rest form) collect (|mkConstructor| x)))))))
```

———-

6.1.61 defun unloadOneConstructor

[remprop p??]
[mkAutoLoad p??]

— defun unloadOneConstructor —

```
(defun |unloadOneConstructor| (cnam fn)
 (remprop cnam 'loaded)
 (setf (symbol-function cnam) (|mkAutoLoad| fn cnam)))
```

———-

6.1.62 defun lisplibDoRename

[replaceFile p??]
[$spadLibFT p??]

— defun lisplibDoRename —

```
(defun |lisplibDoRename| (libName)
 (declare (special |$spadLibFT|))
 (replaceFile (list libName |$spadLibFT| 'a) (list libName 'errorlib 'a)))
```

———-

6.1.63 defun initializeLisplib

[erase p??]
[writeLib1 p193]
[addoptions p??]
[pathnameTypeId p??]
[LAM,FILEACTQ p??]
[$erase p??]
[$libFile p??]
[$libFile p??]
[$lisplibForm p??]

[$lisplibModemap p??]
[$lisplibKind p??]
[$lisplibModemapAlist p??]
[$lisplibAbbreviation p??]
[$lisplibAncestors p??]
[$lisplibOpAlist p??]
[$lisplibOperationAlist p??]
[$lisplibSuperDomain p??]
[$lisplibVariableAlist p??]
[$lisplibSignatureAlist p??]
[/editfile p??]
[/major-version p??]
[errors p??]

— defun initializeLisplib —

```
(defun |initializeLisplib| (libName)
  (declare (special $erase |$libFile| |$lisplibForm|
                    |$lisplibModemap| |$lisplibKind| |$lisplibModemapAlist|
                    |$lisplibAbbreviation| |$lisplibAncestors|
                    |$lisplibOpAlist| |$lisplibOperationAlist|
                    |$lisplibSuperDomain| |$lisplibVariableAlist| errors
                    |$lisplibSignatureAlist| /editfile /major-version errors))
   ($erase libName 'errorlib 'a)
   (setq errors 0)
   (setq |$libFile| (|writeLib1| libname 'errorlib 'a))
   (addoptions 'file |$libFile|)
   (setq |$lisplibForm| nil)
   (setq |$lisplibModemap| nil)
   (setq |$lisplibKind| nil)
   (setq |$lisplibModemapAlist| nil)
   (setq |$lisplibAbbreviation| nil)
   (setq |$lisplibAncestors| nil)
   (setq |$lisplibOpAlist| nil)
   (setq |$lisplibOperationAlist| nil)
   (setq |$lisplibSuperDomain| nil)
   (setq |$lisplibVariableAlist| nil)
   (setq |$lisplibSignatureAlist| nil)
   (when (eq (|pathnameTypeId| /editfile) 'spad)
     (|LAM,FILEACTQ| 'version (list '/versioncheck /major-version))))
```

———-

6.1.64 defun writeLib1

[rdefiostream p??]

— defun writeLib1 —

```
(defun |writeLib1| (fn ft fm)
  (rdefiostream (cons (list 'file fn ft fm) (list '(mode . output)))))
```

———-

6.1.65 defun finalizeLisplib

[lisplibWrite p199]
[removeZeroOne p??]
[namestring p??]
[getConstructorOpsAndAtts p195]
[NRTgenInitialAttributeAlist p??]
[mergeSignatureAndLocalVarAlists p199]
[finalizeDocumentation p444]
[profileWrite p??]
[sayMSG p??]
[$lisplibForm p??]
[$libFile p??]
[$lisplibKind p??]
[$lisplibModemap p??]
[$lisplibCategory p??]
[$/editfile p??]
[$lisplibModemapAlist p??]
[$lisplibForm p??]
[$lisplibModemap p??]
[$FormalMapVariableList p249]
[$lisplibSuperDomain p??]
[$lisplibSignatureAlist p??]
[$lisplibVariableAlist p??]
[$lisplibAttributes p??]
[$lisplibPredicates p??]
[$lisplibAbbreviation p??]
[$lisplibParents p??]
[$lisplibAncestors p??]
[$lisplibSlot1 p??]
[$profileCompiler p??]
[$spadLibFT p??]
[$lisplibCategory p??]
[$pairlis p??]
[$NRTslot1PredicateList p??]

— defun finalizeLisplib —

```
(defun |finalizeLisplib| (libName)
 (let (|$pairlis| |$NRTslot1PredicateList| kind opsAndAtts)
 (declare (special |$pairlis| |$NRTslot1PredicateList| |$spadLibFT|
                   |$lisplibForm| |$profileCompiler| |$libFile|
                   |$lisplibSlot1| |$lisplibAncestors| |$lisplibParents|
                   |$lisplibAbbreviation| |$lisplibPredicates|
                   |$lisplibAttributes| |$lisplibVariableAlist|
                   |$lisplibSignatureAlist| |$lisplibSuperDomain|
                   |$FormalMapVariableList| |$lisplibModemap|
                   |$lisplibModemapAlist| /editfile |$lisplibCategory|
                   |$lisplibKind| errors))
```

```
(|lisplibWrite| "constructorForm"
  (|removeZeroOne| |$lisplibForm|) |$libFile|)
(|lisplibWrite| "constructorKind"
  (setq kind (|removeZeroOne| |$lisplibKind|)) |$libFile|)
(|lisplibWrite| "constructorModemap"
  (|removeZeroOne| |$lisplibModemap|) |$libFile|)
(setq |$lisplibCategory| (or |$lisplibCategory| (cadar |$lisplibModemap|)))
(|lisplibWrite| "constructorCategory" |$lisplibCategory| |$libFile|)
(|lisplibWrite| "sourceFile" (|namestring| /editfile) |$libFile|)
(|lisplibWrite| "modemaps"
  (|removeZeroOne| |$lisplibModemapAlist|) |$libFile|)
(setq opsAndAtts
  (|getConstructorOpsAndAtts| |$lisplibForm| kind |$lisplibModemap|))
(|lisplibWrite| "operationAlist"
  (|removeZeroOne| (car opsAndAtts)) |$libFile|)
(when (eq kind '|category|)
  (setq |$pairlis|
    (loop for a in (rest |$lisplibForm|)
          for v in |$FormalMapVariableList|
      collect (cons a v)))
  (setq |$NRTslot1PredicateList| nil)
  (|NRTgenInitialAttributeAlist| (cdr opsAndAtts)))
(|lisplibWrite| "superDomain"
  (|removeZeroOne| |$lisplibSuperDomain|) |$libFile|)
(|lisplibWrite| "signaturesAndLocals"
  (|removeZeroOne|
   (|mergeSignatureAndLocalVarAlists| |$lisplibSignatureAlist|
                                      |$lisplibVariableAlist|))
      |$libFile|)
(|lisplibWrite| "attributes"
  (|removeZeroOne| |$lisplibAttributes|) |$libFile|)
(|lisplibWrite| "predicates"
  (|removeZeroOne| |$lisplibPredicates|) |$libFile|)
(|lisplibWrite| "abbreviation" |$lisplibAbbreviation| |$libFile|)
(|lisplibWrite| "parents" (|removeZeroOne| |$lisplibParents|) |$libFile|)
(|lisplibWrite| "ancestors" (|removeZeroOne| |$lisplibAncestors|) |$libFile|)
(|lisplibWrite| "documentation" (|finalizeDocumentation|) |$libFile|)
(|lisplibWrite| "slot1Info" (|removeZeroOne| |$lisplibSlot1|) |$libFile|)
(when |$profileCompiler| (|profileWrite|))
(when (and |$lisplibForm| (null (cdr |$lisplibForm|)))
  (setf (get (car |$lisplibForm|) 'niladic) t))
(unless (eql errors 0)
  (|sayMSG| (list "   Errors in processing " kind " " libName ":"))
  (|sayMSG| (list "     not replacing " |$spadLibFT| " for" libName)))))
```

———-

6.1.66 defun getConstructorOpsAndAtts

[getCategoryOpsAndAtts p196]
[getFunctorOpsAndAtts p198]

— defun getConstructorOpsAndAtts —

```
(defun |getConstructorOpsAndAtts| (form kind modemap)
 (if (eq kind '|category|)
  (|getCategoryOpsAndAtts| form)
  (|getFunctorOpsAndAtts| form modemap)))
```

———-

6.1.67 defun getCategoryOpsAndAtts

[transformOperationAlist p196]
[getSlotFromCategoryForm p196]
[getSlotFromCategoryForm p196]

— defun getCategoryOpsAndAtts —

```
(defun |getCategoryOpsAndAtts| (catForm)
 (cons (|transformOperationAlist| (|getSlotFromCategoryForm| catForm 1))
       (|getSlotFromCategoryForm| catForm 2)))
```

———-

6.1.68 defun getSlotFromCategoryForm

[eval p??]
[take p??]
[systemErrorHere p??]
[$FormalMapVariableList p249]

— defun getSlotFromCategoryForm —

```
(defun |getSlotFromCategoryForm| (opargs index)
 (let (op argl u)
 (declare (special |$FormalMapVariableList|))
  (setq op (first opargs))
  (setq argl (rest opargs))
  (setq u
   (|eval| (cons op (mapcar 'mkq (take (|#| argl) |$FormalMapVariableList|)))))
  (if (null (vecp u))
    (|systemErrorHere| "getSlotFromCategoryForm")
    (elt u index))))
```

———-

6.1.69 defun transformOperationAlist

This transforms the operationAlist which is written out onto LISPLIBs. The original form of this list is a list of items of the form:

```
((<op> <signature>) (<condition> (ELT $ n)))
```

The new form is an op-Alist which has entries

```
(<op> . signature-Alist)
```

where signature-Alist has entries

```
(<signature> . item)
```

where item has form

```
(<slotNumber> <condition> <kind>)
```

```
where <kind> =
   NIL  => function
  CONST => constant ... and others
```

[member p??]
[keyedSystemError p??]
[assoc p??]
[lassq p??]
[insertAlist p??]
[$functionLocations p??]

— defun transformOperationAlist —

```
(defun |transformOperationAlist| (operationAlist)
 (let (op sig condition implementation eltEtc impOp kind u n signatureItem
       itemList newAlist)
 (declare (special |$functionLocations|))
  (setq newAlist nil)
  (dolist (item operationAlist)
   (setq op (caar item))
   (setq sig (cadar item))
   (setq condition (cadr item))
   (setq implementation (caddr item))
   (setq kind
    (cond
     ((and (consp implementation) (consp (qrest implementation))
           (consp (qcddr implementation))
           (eq (qcdddr implementation) nil)
           (progn (setq n (qthird implementation)) t)
           (|member| (setq eltEtc (qfirst implementation)) '(const elt)))
       eltEtc)
     ((consp implementation)
       (setq impOp (qfirst implementation))
       (cond
        ((eq impop 'xlam) implementation)
        ((|member| impOp '(const |Subsumed|)) impOp)
        (t (|keyedSystemError| "Unexpected type of entry in domain: %1s"
             (list impop)))))
     ((eq implementation '|mkRecord|) '|mkRecord|)
     (t (|keyedSystemError| "Unexpected type of entry in domain: %1s"
         (list implementation)))))
   (when (setq u (|assoc| (list op sig) |$functionLocations|))
     (setq n (cons n (cdr u))))
   (setq signatureItem
     (if (eq kind 'elt)
       (if (eq condition t)
```

```
        (list sig n)
        (list sig n condition))
      (list sig n condition kind)))
  (setq itemList (cons signatureItem (lassq op newAlist)))
  (setq newAlist (|insertAlist| op itemList newAlist)))
 newAlist))
```

———-

6.1.70 defun getFunctorOpsAndAtts

[transformOperationAlist p196]
[getSlotFromFunctor p198]

— defun getFunctorOpsAndAtts —

```
(defun |getFunctorOpsAndAtts| (form modemap)
 (cons (|transformOperationAlist| (|getSlotFromFunctor| form 1 modemap))
      (|getSlotFromFunctor| form 2 modemap)))
```

———-

6.1.71 defun getSlotFromFunctor

[compMakeCategoryObject p198]
[systemErrorHere p??]
[$e p??]
[$lisplibOperationAlist p??]

— defun getSlotFromFunctor —

```
(defun |getSlotFromFunctor| (arg1 slot arg2)
 (declare (ignore arg1))
 (let (tt)
 (declare (special |$e| |$lisplibOperationAlist|))
  (cond
   ((eql slot 1) |$lisplibOperationAlist|)
   (t
    (setq tt (or (|compMakeCategoryObject| (cadar arg2) |$e|)
                 (|systemErrorHere| "getSlotFromFunctor")))
    (elt (car tt) slot)))))
```

———-

6.1.72 defun compMakeCategoryObject

[isCategoryForm p??]
[mkEvalableCategoryForm p171]
[$e p??]
[$Category p??]

— defun compMakeCategoryObject —

```
(defun |compMakeCategoryObject| (c |$e|)
 (declare (special |$e|))
 (let (u)
 (declare (special |$Category|))
  (cond
   ((null (|isCategoryForm| c |$e|)) nil)
   ((setq u (|mkEvalableCategoryForm| c)) (list (|eval| u) |$Category| |$e|))
   (t nil))))
```

———-

6.1.73 defun mergeSignatureAndLocalVarAlists

[lassoc p??]

— defun mergeSignatureAndLocalVarAlists —

```
(defun |mergeSignatureAndLocalVarAlists| (signatureAlist localVarAlist)
 (loop for item in signatureAlist
  collect
   (cons (first item)
    (cons (rest item)
     (lassoc (first item) localVarAlist)))))
```

———-

6.1.74 defun lisplibWrite

[rwrite128 p??]
[$lisplib p??]

— defun lisplibWrite —

```
(defun |lisplibWrite| (prop val filename)
 (declare (special $lisplib))
 (when $lisplib (|rwrite| prop val filename)))
```

———-

6.1.75 defun isCategoryPackageName

[pname p??]
[maxindex p??]
[char p??]

— defun isCategoryPackageName —

```
(defun |isCategoryPackageName| (nam)
```

```
 (let (p)
  (setq p (pname (|opOf| nam)))
  (equal (elt p (maxindex p)) #\&)))
```

———-

6.1.76 defun NRTgetLookupFunction

Compute the lookup function (complete or incomplete) [sublis p??]
[NRTextendsCategory1 p??]
[getExportCategory p??]
[sayBrightly p??]
[sayBrightlyNT p??]
[bright p??]
[form2String p??]
[$why p??]
[$why p??]
[$pairlis p??]

— defun NRTgetLookupFunction —

```
(defun |NRTgetLookupFunction| (domform exCategory addForm)
 (let (|$why| extends u msg v)
 (declare (special |$why| |$pairlis|))
  (setq domform (sublis |$pairlis| domform))
  (setq addForm (sublis |$pairlis| addForm))
  (setq |$why| nil)
  (cond
    ((atom addForm) '|lookupComplete|)
    (t
     (setq extends
      (|NRTextendsCategory1| domform exCategory (|getExportCategory| addForm)))
     (cond
      ((null extends)
        (setq u (car |$why|))
        (setq msg (cadr |$why|))
        (setq v (cddr |$why|))
        (|sayBrightly|
           "--------------non extending category----------------------")
        (|sayBrightlyNT|
         (cons ".."
          (append (|bright| (|form2String| domform)) (list '|of cat |))))
        (print u)
        (|sayBrightlyNT| (|bright| msg))
        (if v (print (car v)) (terpri))))
     (if extends
       '|lookupIncomplete|
       '|lookupComplete|)))))
```

———-

6.1.77 defun NRTgetLocalIndex

[NRTassocIndex p317]
[NRTaddInner p??]
[compOrCroak p528]
[rplaca p??]
[$NRTaddForm p??]
[$formalArgList p??]
[$NRTdeltaList p??]
[$NRTdeltaListComp p??]
[$NRTdeltaLength p??]
[$NRTbase p??]
[$EmptyMode p166]
[$e p??]

— defun NRTgetLocalIndex —

```
(defun |NRTgetLocalIndex| (item)
 (let (k value saveNRTdeltaListComp saveIndex compEntry)
 (declare (special |$e| |$EmptyMode| |$NRTdeltaLength| |$NRTbase|
                   |$NRTdeltaListComp| |$NRTdeltaList| |$formalArgList|
                   |$NRTaddForm|))
   (cond
     ((setq k (|NRTassocIndex| item)) k)
     ((equal item |$NRTaddForm|) 5)
     ((eq item '$) 0)
     ((eq item '$$) 2)
     (t
       (when (member item |$formalArgList|) (setq value item))
       (cond
         ((and (atom item) (null (member item '($ $$))) (null value))
           (setq |$NRTdeltaList|
             (cons (cons '|domain| (cons (|NRTaddInner| item) value))
                   |$NRTdeltaList|))
           (setq |$NRTdeltaListComp| (cons item |$NRTdeltaListComp|))
           (setq |$NRTdeltaLength| (1+ |$NRTdeltaLength|))
           (1- (+ |$NRTbase| |$NRTdeltaLength|)))
         (t
          (setq |$NRTdeltaList|
           (cons (cons '|domain| (cons (|NRTaddInner| item) value))
                 |$NRTdeltaList|))
          (setq saveNRTdeltaListComp
            (setq |$NRTdeltaListComp| (cons nil |$NRTdeltaListComp|)))
          (setq saveIndex (+ |$NRTbase| |$NRTdeltaLength|))
          (setq |$NRTdeltaLength| (1+ |$NRTdeltaLength|))
          (setq compEntry (car (|compOrCroak| item |$EmptyMode| |$e|)))
          (rplaca saveNRTdeltaListComp compEntry)
          saveIndex))))))
```

———-

6.1.78 defun augmentLisplibModemapsFromFunctor

[formal2Pattern p203]
[mkAlistOfExplicitCategoryOps p181]
[allLASSOCs p203]
[member p??]
[mkDatabasePred p203]
[mkpf p??]
[listOfPatternIds p??]
[interactiveModemapForm p183]
[$lisplibModemapAlist p??]
[$PatternVariableList p??]
[$e p??]
[$lisplibModemapAlist p??]
[$e p??]

— defun augmentLisplibModemapsFromFunctor —

```
(defun |augmentLisplibModemapsFromFunctor| (form opAlist signature)
 (let (argl nonCategorySigAlist op pred sel predList sig predp z skip modemap)
 (declare (special |$lisplibModemapAlist| |$PatternVariableList| |$e|))
  (setq form (|formal2Pattern| form))
  (setq argl (cdr form))
  (setq opAlist (|formal2Pattern| opAlist))
  (setq signature (|formal2Pattern| signature))
  ; We are going to be EVALing categories containing these pattern variables
  (loop for u in form for v in signature
   do (when (member u |$PatternVariableList|)
       (setq |$e| (|put| u '|mode| v |$e|))))
  (when
   (setq nonCategorySigAlist (|mkAlistOfExplicitCategoryOps| (CAR signature)))
   (loop for entry in opAlist
    do
     (setq op (caar entry))
     (setq sig (cadar entry))
     (setq pred (cadr entry))
     (setq sel (caddr entry))
     (when
      (let (result)
       (loop for catSig in (|allLASSOCs| op nonCategorySigAlist)
        do (setq result (or result  (|member| sig catSig))))
       result)
     (setq skip (when (and argl (contained '$ (cdr sig))) 'skip))
     (setq sel (subst form '$ sel :test #'equal))
     (setq predList
      (loop for a in argl for m in (rest signature)
       when (|member| a |$PatternVariableList|)
       collect (list a m)))
     (setq sig (subst form '$ sig :test #'equal))
     (setq predp
      (mkpf
       (cons pred (loop for y in predList collect (|mkDatabasePred| y)))
       'and))
```

```
    (setq z (|listOfPatternIds| predList))
    (when (some #'(lambda (u) (null (member u z))) argl)
      (|sayMSG| (list "cannot handle modemap for " op "by pattern match"))
      (setq skip 'skip))
    (setq modemap (list (cons form sig) (cons predp (cons sel skip))))
    (setq |$lisplibModemapAlist|
     (cons
      (cons op (|interactiveModemapForm| modemap))
      |$lisplibModemapAlist|)))))))
```

———-

6.1.79 defun allLASSOCs

— defun allLASSOCs —

```
(defun |allLASSOCs| (op alist)
 (loop for value in alist
  when (equal (car value) op)
  collect value))
```

———-

6.1.80 defun formal2Pattern

[sublis p??]
[pairList p??]
[$PatternVariableList p??]

— defun formal2Pattern —

```
(defun |formal2Pattern| (x)
 (declare (special |$PatternVariableList|))
 (sublis (pairList |$FormalMapVariableList| (cdr |$PatternVariableList|)) x))
```

———-

6.1.81 defun mkDatabasePred

[isCategoryForm p??]
[$e p??]

— defun mkDatabasePred —

```
(defun |mkDatabasePred| (arg)
 (let (a z)
 (declare (special |$e|))
  (setq a (car arg))
  (setq z (cadr arg))
  (if (|isCategoryForm| z |$e|)
```

```
    (list '|ofCategory| a z)
    (list '|ofType| a z))))
```

———–

6.1.82 defun disallowNilAttribute

— defun disallowNilAttribute —

```
(defun |disallowNilAttribute| (x)
 (loop for y in x when (and (car y) (not (eq (car y) '|nil|)))
  collect y))
```

———–

6.1.83 defun bootStrapError

[mkq p??]
[namestring p??]
[mkDomainConstructor p??]

— defun bootStrapError —

```
(defun |bootStrapError| (functorForm sourceFile)
 (list 'cond
  (list '|$bootStrapMode|
   (list 'vector (|mkDomainConstructor| functorForm) nil nil nil nil nil))
  (list ''t
   (list '|systemError|
    (list 'list (MKQ (CAR functorForm)) "from"
          (mkq (|namestring| sourceFile)) "needs to be compiled")))))
```

———–

6.1.84 defun reportOnFunctorCompilation

[displayMissingFunctions p205]
[sayBrightly p??]
[displaySemanticErrors p??]
[displayWarnings p??]
[addStats p??]
[normalizeStatAndStringify p??]
[$op p??]
[$functorStats p??]
[$functionStats p??]
[$warningStack p??]
[$semanticErrorStack p??]

— defun reportOnFunctorCompilation —

```
(defun |reportOnFunctorCompilation| ()
 (declare (special |$op| |$functorStats| |$functionStats|
                   |$warningStack| |$semanticErrorStack|))
   (|displayMissingFunctions|)
   (when |$semanticErrorStack| (|sayBrightly| " "))
   (|displaySemanticErrors|)
   (when |$warningStack| (|sayBrightly| " "))
   (|displayWarnings|)
   (setq |$functorStats| (|addStats| |$functorStats| |$functionStats|))
   (|sayBrightly|
     (cons '|%l|
      (append (|bright| "  Cumulative Statistics for Constructor")
       (list |$op|))))
   (|sayBrightly|
    (cons "      Time:"
     (append (|bright| (|normalizeStatAndStringify| (second |$functorStats|)))
       (list "seconds"))))
   (|sayBrightly| " ")
   '|done|)
```

———

6.1.85 defun displayMissingFunctions

[member p??]
[getmode p??]
[sayBrightly p??]
[bright p??]
[formatUnabbreviatedSig p??]
[$env p??]
[$formalArgList p??]
[$CheckVectorList p??]

— defun displayMissingFunctions —

```
(defun |displayMissingFunctions| ()
 (let (i loc exp)
 (declare (special |$env| |$formalArgList| |$CheckVectorList|))
 (unless |$CheckVectorList|
  (setq loc nil)
  (setq exp nil)
  (loop for cvl in |$CheckVectorList| do
   (unless (cdr cvl)
    (if (and (null (|member| (caar cvl) |$formalArgList|))
             (consp (|getmode| (caar cvl) |$env|))
             (eq (qfirst (|getmode| (caar cvl) |$env|)) '|Mapping|))
      (push (list (caar cvl) (cadar cvl)) loc)
      (push (list (caar cvl) (cadar cvl)) exp))))
  (when loc
   (|sayBrightly| (cons '|%l| (|bright| "  Missing Local Functions:")))
   (setq i 0)
```

```
   (loop for item in loc do
    (|sayBrightly|
     (cons "      [" (cons (incf i) (cons "]"
      (append (|bright| (first item))
       (cons '|: | (|formatUnabbreviatedSig| (second item))))))))))
 (when exp
  (|sayBrightly| (cons '|%l| (|bright| "  Missing Exported Functions:")))
  (setq i 0)
  (loop for item in exp do
   (|sayBrightly|
    (cons "      [" (cons (incf i) (cons "]"
     (append (|bright| (first item))
      (cons '|: | (|formatUnabbreviatedSig| (second item))))))))))))
```

———-

6.1.86 defun makeFunctorArgumentParameters

[assq p??]
[isCategoryForm p??]
[genDomainViewList0 p208]
[union p??]
[$ConditionalOperators p??]
[$alternateViewList p??]
[$forceAdd p??]

— defun makeFunctorArgumentParameters —

```
(defun |makeFunctorArgumentParameters| (argl sigl target)
 (labels (
  (augmentSig (s ss)
   (let (u)
   (declare (special |$ConditionalOperators|))
    (if ss
     (progn
      (loop for u in ss do (push (rest u) |$ConditionalOperators|))
      (if (and (consp s) (eq (qfirst s) '|Join|))
       (progn
        (if (setq u (assq 'category ss))
         (subst (append u ss) u s :test #'equal)
         (cons '|Join|
          (append (rest s) (list (cons 'category (cons '|package| ss)))))))
       (list '|Join| s (cons 'category (cons '|package| ss)))))
     s)))
  (fn (a s)
   (declare (special |$CategoryFrame|))
    (if (|isCategoryForm| s |$CategoryFrame|)
     (if (and (consp s) (eq (qfirst s) '|Join|))
      (|genDomainViewList0| a (rest s))
      (list (|genDomainView| a s '|getDomainView|)))
     (list a)))
  (findExtras (a target)
```

```
  (cond
   ((and (consp target) (eq (qfirst target) '|Join|))
    (reduce #'|union|
     (loop for x in (qrest target)
       collect (findExtras a x))))
   ((and (consp target) (eq (qfirst target) 'category))
    (reduce #'|union|
     (loop for x in (qcddr target)
      collect (findExtras1 a x))))))
 (findExtras1 (a x)
  (cond
   ((and (consp x) (or (eq (qfirst x) 'and)) (eq (qfirst x) 'or))
     (reduce #'|union|
       (loop for y in (rest x) collect (findExtras1 a y))))
   ((and (consp x) (eq (qfirst x) 'if)
         (consp (qrest x)) (consp (qcddr x))
         (consp (qcdddr x))
         (eq (qcddddr x) nil))
     (|union| (findExtrasP a (second x))
              (|union|
               (findExtras1 a (third x))
               (findExtras1 a (fourth x)))))))
 (findExtrasP (a x)
  (cond
   ((and (consp x) (or (eq (qfirst x) 'and)) (eq (qfirst x) 'or))
     (reduce #'|union|
       (loop for y in (rest x) collect (findExtrasP a y))))
   ((and (consp x) (eq (qfirst x) '|has|)
         (consp (qrest x)) (consp (qcddr x))
         (consp (qcdddr x))
         (eq (qcddddr x) nil))
     (|union| (findExtrasP a (second x))
              (|union|
               (findExtras1 a (third x))
               (findExtras1 a (fourth x)))))
   ((and (consp x) (eq (qfirst x) '|has|)
         (consp (qrest x)) (equal (qsecond x) a)
         (consp (qcddr x))
         (eq (qcdddr x) nil)
         (consp (qthird x))
         (eq (qcaaddr x) 'signature))
     (list (third x)))))

)
(let (|$alternateViewList| |$forceAdd| |$ConditionalOperators|)
(declare (special |$alternateViewList| |$forceAdd| |$ConditionalOperators|))
 (setq |$alternateViewList| nil)
 (setq |$forceAdd| t)
 (setq |$ConditionalOperators| nil)
 (mapcar #'reduce
  (loop for a in argl for s in sigl do
    (fn a (augmentSig s (findExtras a target)))))))
```

———-

6.1.87 defun genDomainViewList0

[getDomainViewList p??]

— defun genDomainViewList0 —

```
(defun |genDomainViewList0| (id catlist)
 (|genDomainViewList| id catlist t))
```

———-

6.1.88 defun genDomainViewList

[isCategoryForm p??]
[genDomainView p208]
[genDomainViewList p208]
[$EmptyEnvironment p??]

— defun genDomainViewList —

```
(defun |genDomainViewList| (id catlist firsttime)
 (declare (special |$EmptyEnvironment|) (ignore firsttime))
  (cond
   ((null catlist) nil)
   ((and (consp catlist) (eq (qrest catlist) nil)
         (null (|isCategoryForm| (first catlist) |$EmptyEnvironment|)))
      nil)
   (t
    (cons
     (|genDomainView| id (first catlist) '|getDomainView|)
     (|genDomainViewList| id (rest catlist) nil)))))
```

———-

6.1.89 defun genDomainView

[genDomainOps p209]
[augModemapsFromCategory p244]
[mkDomainConstructor p??]
[member p??]
[$e p??]
[$getDomainCode p??]

— defun genDomainView —

```
(defun |genDomainView| (name c viewSelector)
 (let (code cd)
 (declare (special |$getDomainCode| |$e|))
 (cond
```

```
  ((and (consp c) (eq (qfirst c) 'category) (consp (qrest c)))
    (|genDomainOps| name name c))
   (t
    (setq code
     (if (and (consp c) (eq (qfirst c) '|SubsetCategory|)
              (consp (qrest c)) (consp (qcddr c))
              (eq (qcdddr c) nil))
      (second c)
      c))
    (setq |$e| (|augModemapsFromCategory| name nil c |$e|))
    (setq cd
     (list 'let name (list viewSelector name (|mkDomainConstructor| code))))
    (unless (|member| cd |$getDomainCode|)
      (setq |$getDomainCode| (cons cd |$getDomainCode|)))
    name))))
```

6.1.90 defun genDomainOps

[getOperationAlist p249]
[substNames p250]
[mkq p??]
[mkDomainConstructor p??]
[addModemap p252]
[$e p??]
[$ConditionalOperators p??]
[$getDomainCode p??]

— defun genDomainOps —

```
(defun |genDomainOps| (viewName dom cat)
 (let (siglist oplist cd i)
 (declare (special |$e| |$ConditionalOperators| |$getDomainCode|))
  (setq oplist (|getOperationAlist| dom dom cat))
  (setq siglist (loop for lst in oplist collect (first lst)))
  (setq oplist (|substNames| dom viewName dom oplist))
  (setq cd
   (list 'let viewName
    (list '|mkOpVec| dom
     (cons 'list
      (loop for opsig in siglist
       collect
        (list 'list (mkq (first opsig))
         (cons 'list
          (loop for mode in (rest opsig)
           collect (|mkDomainConstructor| mode)))))))))
  (setq |$getDomainCode| (cons cd |$getDomainCode|))
  (setq i 0)
  (loop for item in oplist do
   (if (|member| (first item) |$ConditionalOperators|)
    (setq |$e| (|addModemap| (caar item) dom (cadar item) nil
```

```
            (list 'elt viewName (incf i)) |$e|))
   (setq |$e| (|addModemap| (caar item) dom (cadar item) (second item)
            (list 'elt viewName (incf i)) |$e|))))
 viewName))
```

———-

6.1.91 defun mkOpVec

[getPrincipalView p??]
[getOperationAlistFromLisplib p??]
[opOf p??]
[length p??]
[assq p??]
[assoc p??]
[sublis p??]
[AssocBarGensym p211]
[$FormalMapVariableList p249]
[Undef p??]

— defun mkOpVec —

```
(defun |mkOpVec| (dom siglist)
 (let (substargs oplist ops u noplist i tmp1)
 (declare (special |$FormalMapVariableList| |Undef|))
  (setq dom (|getPrincipalView| dom))
  (setq substargs
    (cons (cons '$ (elt dom 0))
          (loop for a in |$FormalMapVariableList| for x in (rest (elt dom 0))
           collect (cons a x))))
  (setq oplist (|getOperationAlistFromLisplib| (|opOf| (elt dom 0))))
  (setq ops (make-array (|#| siglist)))
  (setq i -1)
  (loop for opSig in siglist do
    (incf i)
    (setq u (assq (first opSig) oplist))
    (setq tmp1 (|assoc| (second opSig) u))
    (cond
     ((and (consp tmp1) (consp (qrest tmp1))
           (consp (qcddr tmp1)) (consp (qcdddr tmp1))
           (eq (qcddddr tmp1) nil)
           (eq (qfourth tmp1) 'elt))
      (setelt ops i (elt dom (second tmp1))))
     (t
      (setq noplist (sublis substargs u))
      (setq tmp1
        (|AssocBarGensym|
          (subst (elt dom 0) '$ (second opSig) :test #'equal) noplist))
      (cond
       ((and (consp tmp1) (consp (qrest tmp1)) (consp (qcddr tmp1))
             (consp (qcdddr tmp1))
             (eq (qcddddr tmp1) nil)
```

```
             (eq (qfourth tmp1) 'elt))
         (setelt ops i (elt dom (second tmp1))))
       (t
         (setelt ops i (cons |Undef| (cons (list (elt dom 0) i) opSig))))))))
  ops))
```

———-

6.1.92 defun AssocBarGensym

[EqualBarGensym p230]

— defun AssocBarGensym —

```
(defun |AssocBarGensym| (key z)
 (loop for x in z
  do (when (and (consp x) (|EqualBarGensym| key (car x))) (return x))))
```

———-

6.1.93 defun orderByDependency

[say p??]
[userError p??]
[intersection p??]
[member p??]
[remdup p??]

— defun orderByDependency —

```
(defun |orderByDependency| (vl dl)
 (let (selfDependents fatalError newl orderedVarList vlp dlp)
  (setq selfDependents
   (loop for v in vl for d in dl
    when (member v d)
    collect v))
  (loop for v in vl for d in dl
   when (member v d)
   do (say v "depends on itself")
      (setq fatalError t))
  (cond
    (fatalError (|userError| "Parameter specification error"))
    (t
     (loop until (null vl) do
       (setq newl
         (loop for v in vl for d in dl
          when (null (|intersection| d vl))
          collect v))
        (if (null newl)
         (setq vl nil) ; force loop exit
         (progn
          (setq orderedVarList (append newl orderedVarList))
```

```
          (setq vlp (setdifference vl newl))
          (setq dlp
          (loop for x in vl for d in dl
           when (|member| x vlp)
           collect (setdifference d newl)))
          (setq vl vlp)
          (setq dl dlp))))
        (when (and newl orderedVarList) (remdup (nreverse orderedVarList)))))))
```

——————

6.2 Code optimization routines

6.2.1 defun optimizeFunctionDef

[rplac p**??**]
[sayBrightlyI p**??**]
[optimize p213]
[pp p**??**]
[bright p**??**]
[$reportOptimization p**??**]

— **defun optimizeFunctionDef** —

```
(defun |optimizeFunctionDef| (def)
 (labels (
  (fn (x g)
    (cond
     ((and (consp x) (eq (qfirst x) 'throw) (consp (qrest x))
           (equal (qsecond x) g))
       (|rplac| (car x) 'return)
       (|rplac| (cdr x)
        (replaceThrowByReturn (qcddr x) g)))
     ((atom x) nil)
     (t
      (replaceThrowByReturn (car x) g)
      (replaceThrowByReturn (cdr x) g))))
  (replaceThrowByReturn (x g)
   (fn x g)
   x)
  (removeTopLevelCatch (body)
   (if (and (consp body) (eq (qfirst body) 'catch) (consp (qrest body))
            (consp (qcddr body)) (eq (qcdddr body) nil))
    (removeTopLevelCatch
      (replaceThrowByReturn
        (qthird body) (qsecond body)))
    body)))
 (let (defp name slamOrLam args body bodyp)
 (declare (special |$reportOptimization|))
  (when |$reportOptimization|
    (|sayBrightlyI| (|bright| "Original LISP code:"))
```

```
    (|pp| def))
  (setq defp (|optimize| (copy def)))
  (when |$reportOptimization|
    (|sayBrightlyI| (|bright| "Optimized LISP code:"))
    (|pp| defp)
    (|sayBrightlyI| (|bright| "Final LISP code:")))
  (setq name (car defp))
  (setq slamOrLam (caadr defp))
  (setq args (cadadr defp))
  (setq body (car (cddadr defp)))
  (setq bodyp (removeTopLevelCatch body))
  (list name (list slamOrLam args bodyp)))))
```

———-

6.2.2 defun optimize

[optimize p213]
[say p??]
[prettyprint p??]
[rplac p??]
[optIF2COND p215]
[getl p??]
[subrname p216]

— defun optimize —

```
(defun |optimize| (x)
 (labels (
  (opt (x)
   (let (argl body a y op)
    (cond
     ((atom x) nil)
     ((eq (setq y (car x)) 'quote) nil)
     ((eq y 'closedfn) nil)
     ((and (consp y) (consp (qfirst y)) (eq (qcaar y) 'xlam)
           (consp (qcdar y)) (consp (qcddar y))
           (eq (qcdddar y) nil))
      (setq argl (qcadar y))
      (setq body (qcaddar y))
      (setq a (qrest y))
      (|optimize| (cdr x))
      (cond
       ((eq argl '|ignore|) (rplac (car x) body))
       (t
         (when (null (<= (length argl) (length a)))
           (say "length mismatch in XLAM expression")
           (prettyprint y))
          (rplac (car x)
           (|optimize|
            (|optXLAMCond|
             (sublis (pairList argl a) body)))))))
```

```
  ((atom y)
    (|optimize| (cdr x))
    (cond
     ((eq y '|true|) (rplac (car x) '''T))
     ((eq y '|false|) (rplac (car x) nil))))
  ((eq (car y) 'if)
    (rplac (car x) (|optIF2COND| y))
    (setq y (car x))
    (when (setq op (getl (|subname| (car y)) 'optimize))
     (|optimize| (cdr x))
     (rplac (car x) (funcall op (|optimize| (car x))))))
  ((setq op (getl (|subname| (car y)) 'optimize))
     (|optimize| (cdr x))
     (rplac (car x) (funcall op (|optimize| (car x)))))
  (t
    (rplac (car x) (|optimize| (car x)))
    (|optimize| (cdr x))))))
 (opt x)
 x))
```

———-

6.2.3 defun optXLAMCond

[optCONDtail p214]
[optPredicateIfTrue p215]
[optXLAMCond p214]
[rplac p??]

— defun optXLAMCond —

```
(defun |optXLAMCond| (x)
 (cond
  ((and (consp x) (eq (qfirst x) 'cond) (consp (qrest x))
        (consp (qsecond x)) (consp (qcdadr x))
        (eq (qcddadr x) nil))
    (if (|optPredicateIfTrue| (qcaadr x))
      (qcadadr x)
      (cons 'cond (cons (qsecond x) (|optCONDtail| (qcddr x))))))
  ((atom x) x)
  (t
    (rplac (car x) (|optXLAMCond| (car x)))
    (rplac (cdr x) (|optXLAMCond| (cdr x)))
    x)))
```

———-

6.2.4 defun optCONDtail

[optCONDtail p214]
[$true p??]

— **defun optCONDtail** —

```
(defun |optCONDtail| (z)
 (declare (special |$true|))
 (when z
  (cond
   ((|optPredicateIfTrue| (caar z)) (list (list |$true| (cadar z))))
   ((null (cdr z)) (list (car z) (list |$true| (list '|CondError|))))
   (t (cons (car z) (|optCONDtail| (cdr z)))))))
```

———

6.2.5 defvar $BasicPredicates

If these predicates are found in an expression the code optimizer routine optPredicateIfTrue then optXLAM will replace the call with the argument. This is used for predicates that test the type of their argument so that, for instance, a call to integerp on an integer will be replaced by that integer if it is true. This represents a simple kind of compile-time type evaluation.

— **initvars** —

```
(defvar |$BasicPredicates| '(integerp stringp floatp))
```

———

6.2.6 defun optPredicateIfTrue

[$BasicPredicates p215]

— **defun optPredicateIfTrue** —

```
(defun |optPredicateIfTrue| (p)
 (declare (special |$BasicPredicates|))
  (cond
   ((and (consp p) (eq (qfirst p) 'quote)) T)
   ((and (consp p) (consp (qrest p)) (eq (qcddr p) nil)
      (member (qfirst p) |$BasicPredicates|) (funcall (qfirst p) (qsecond p)))
     t)
   (t nil)))
```

———

6.2.7 defun optIF2COND

[optIF2COND p215]
[$true p??]

— **defun optIF2COND** —

```
(defun |optIF2COND| (arg)
```

```
(let (a b c)
(declare (special |$true|))
 (setq a (cadr arg))
 (setq b (caddr arg))
 (setq c (cadddr arg))
 (cond
  ((eq b '|noBranch|) (list 'cond (list (list 'null a ) c)))
  ((eq c '|noBranch|) (list 'cond (list a b)))
  ((and (consp c) (eq (qfirst c) 'if))
    (cons 'cond (cons (list a b) (cdr (|optIF2COND| c)))))
  ((and (consp c) (eq (qfirst c) 'cond))
    (cons 'cond (cons (list a b) (qrest c))))
  (t
    (list 'cond (list a b) (list |$true| c))))))
```

———

6.2.8 defun subrname

[identp p??]
[compiled-function-p p??]
[mbpip p??]
[bpiname p??]

— defun subrname —

```
(defun |subrname| (u)
 (cond
  ((identp u) u)
  ((or (compiled-function-p u) (mbpip u)) (bpiname u))
  (t nil)))
```

———

6.2.9 Special case optimizers

Optimization functions are called through the OPTIMIZE property on the symbol property list. The current list is:

```
|call|      optCall
seq         optSEQ
eq          optEQ
minus       optMINUS
qsminus     optQSMINUS
-           opt-
lessp       optLESSP
spadcall    optSPADCALL
|           optSuchthat
catch       optCatch
cond        optCond
|mkRecord|  optMkRecord
recordelt   optRECORDELT
```

```
setrecordelt optSETRECORDELT
recordcopy   optRECORDCOPY
```

Be aware that there are case-sensitivity issues. When found in the s-expression, each symbol in the left column will call a custom optimization routine in the right column. The optimization routines are below. Note that each routine has a special chunk in postvars using eval-when to set the property list at load time.

These optimizations are done destructively. That is, they modify the function in-place using rplac.

Not all of the optimization routines are called through the property list. Some are called only from other optimization routines, e.g. optPackageCall.

6.2.10 defplist optCall

— postvars —

```
(eval-when (eval load)
 (setf (get '|call| 'optimize) '|optCall|))
```

————-

6.2.11 defun Optimize "call" expressions

[optimize p213]
[rplac p??]
[optPackageCall p218]
[optCallSpecially p218]
[systemErrorHere p??]
[$QuickCode p??]
[$bootStrapMode p??]

— defun optCall —

```
(defun |optCall| (x)
 (let (u tmp1 fn a name q r n w)
 (declare (special |$QuickCode| |$bootStrapMode|))
   (setq u (cdr x))
   (setq x (|optimize| (list u)))
   (cond
    ((atom (car x)) (car x))
    (t
     (setq tmp1 (car x))
     (setq fn (car tmp1))
     (setq a (cdr tmp1))
     (cond
      ((atom fn) (rplac (cdr x) a) (rplac (car x) fn))
      ((and (consp fn) (eq (qfirst fn) 'pac)) (|optPackageCall| x fn a))
      ((and (consp fn) (eq (qfirst fn) '|applyFun|)
            (consp (qrest fn)) (eq (qcddr fn) nil))
       (setq name (qsecond fn))
```

```
       (rplac (car x) 'spadcall)
       (rplac (cdr x) (append a (cons name nil)))
       x)
      ((and (consp fn) (consp (qrest fn)) (consp (qcddr fn))
            (eq (qcdddr fn) nil)
            (member (qfirst fn) '(elt qrefelt const)))
       (setq q (qfirst fn))
       (setq r (qsecond fn))
       (setq n (qthird fn))
       (cond
        ((and (null |$bootStrapMode|) (setq w (|optCallSpecially| q x n r)))
          w)
        ((eq q 'const)
          (list '|spadConstant| r n))
        (t
          (rplac (car x) 'spadcall)
          (when |$QuickCode| (rplaca fn 'qrefelt))
          (rplac (cdr x) (append a (list fn)))
          x)))
      (t (|systemErrorHere| "optCall")))))))
```

———-

6.2.12 defun optPackageCall

[rplaca p??]
[rplacd p??]

— defun optPackageCall —

```
(defun |optPackageCall| (x arg2 arglist)
 (let (packageVariableOrForm functionName)
  (setq packageVariableOrForm (second arg2))
  (setq functionName (third arg2))
  (rplaca x functionName)
  (rplacd x (append arglist (list packageVariableOrForm)))
  x))
```

———-

6.2.13 defun optCallSpecially

[lassoc p??]
[get p??]
[opOf p??]
[optSpecialCall p219]
[$specialCaseKeyList p??]
[$getDomainCode p??]
[$optimizableConstructorNames p??]
[$e p??]

— defun optCallSpecially —

```
(defun |optCallSpecially| (q x n r)
 (declare (ignore q))
 (labels (
  (lookup (a z)
   (let (zp)
    (when z
     (setq zp (car z))
     (setq z (cdr x))
     (if (and (consp zp) (eq (qfirst zp) 'let) (consp (qrest zp))
              (equal (qsecond zp) a) (consp (qcddr zp)))
      (qthird zp)
      (lookup a z))))))
 (let (tmp1 op y prop yy)
 (declare (special |$specialCaseKeyList| |$getDomainCode| |$e|
                   |$optimizableConstructorNames|))
  (cond
   ((setq y (lassoc r |$specialCaseKeyList|))
     (|optSpecialCall| x y n))
   ((member (ifcar r) |$optimizableConstructorNames|)
     (|optSpecialCall| x r n))
   ((and (setq y (|get| r '|value| |$e|))
         (member (|opOf| (car y)) |$optimizableConstructorNames|))
     (|optSpecialCall| x (car y) n))
   ((and (setq y (lookup r |$getDomainCode|))
         (progn
           (setq tmp1 y)
           (setq op (first tmp1))
           (setq y (second tmp1))
           (setq prop (third tmp1))
          tmp1)
         (setq yy (lassoc y |$specialCaseKeyList|)))
     (|optSpecialCall| x (list op yy prop) n))
    (t nil)))))
```

———

6.2.14 defun optSpecialCall

[optCallEval p221]
[function p??]
[keyedSystemError p??]
[mkq p??]
[getl p??]
[compileTimeBindingOf p220]
[rplac p??]
[optimize p213]
[rplacw p??]
[rplaca p??]
[$QuickCode p??]
[$Undef p??]

— defun optSpecialCall —

```
(defun |optSpecialCall| (x y n)
 (let (yval args tmp1 fn a)
 (declare (special |$QuickCode| |Undef|))
  (setq yval (|optCallEval| y))
  (cond
   ((eq (caaar x) 'const)
     (cond
      ((equal (ifcar (elt yval n)) #'|Undef|)
        (|keyedSystemError|
            "Unexpected error or improper call to system function %1: %2"
          (list "optSpecialCall" "invalid constant")))
      (t (mkq (elt yval n)))))
   ((setq fn (getl (|compileTimeBindingOf| (car (elt yval n))) '|SPADreplace|))
     (|rplac| (cdr x) (cdar x))
     (|rplac| (car x) fn)
     (when (and (consp fn) (eq (qfirst fn) 'xlam))
      (setq x (car (|optimize| (list x)))))
     (if (and (consp x) (eq (qfirst x) 'equal) (progn (setq args (qrest x)) t))
      (rplacw x (def-equal args))
      x))
   (t
    (setq tmp1 (car x))
    (setq fn (car tmp1))
    (setq a (cdr tmp1))
    (rplac (car x) 'spadcall)
    (when |$QuickCode| (rplaca fn 'qrefelt))
    (rplac (cdr x) (append a (list fn)))
     x))))
```

———

6.2.15 defun compileTimeBindingOf

[bpiname p??]
[keyedSystemError p??]
[moan p??]

— defun compileTimeBindingOf —

```
(defun |compileTimeBindingOf| (u)
 (let (name)
  (cond
   ((null (setq name (bpiname u)))
     (|keyedSystemError| "Irregular slot entry: %1s" (list u)))
   ((eq name '|Undef|)
     (moan "optimiser found unknown function"))
   (t name))))
```

———

6.2.16 defun optCallEval

[List p??]
[Integer p??]
[Vector p??]
[PrimititveArray p??]
[FactoredForm p??]
[Matrix p??]
[eval p??]

— defun optCallEval —

```
(defun |optCallEval| (u)
  (cond
    ((and (consp u) (eq (qfirst u) '|List|))
      (|List| (|Integer|)))
    ((and (consp u) (eq (qfirst u) '|Vector|))
      (|Vector| (|Integer|)))
    ((and (consp u) (eq (qfirst u) '|PrimitiveArray|))
      (|PrimitiveArray| (|Integer|)))
    ((and (consp u) (eq (qfirst u) '|FactoredForm|))
     (|FactoredForm| (|Integer|)))
    ((and (consp u) (eq (qfirst u) '|Matrix|))
     (|Matrix| (|Integer|)))
    (t
     (|eval| u))))
```

———-

6.2.17 defplist optSEQ

— postvars —

```
(eval-when (eval load)
 (setf (get 'seq 'optimize) '|optSEQ|))
```

———-

6.2.18 defun optSEQ

— defun optSEQ —

```
(defun |optSEQ| (arg)
 (labels (
  (tryToRemoveSEQ (z)
    (if (and (consp z) (eq (qfirst z) 'seq) (consp (qrest z))
             (eq (qcddr z) nil) (consp (qsecond z))
             (consp (qcdadr z))
             (eq (qcddadr z) nil)
             (member (qcaadr z) '(exit return throw)))
```

```
      (qcadadr z)
      z))
  (SEQToCOND (z)
   (let (transform before aft)
    (setq transform
     (loop for x in z
      while
        (and (consp x) (eq (qfirst x) 'cond) (consp (qrest x))
             (eq (qcddr x) nil) (consp (qsecond x))
             (consp (qcdadr x))
             (eq (qcddadr x) nil)
             (consp (qcadadr x))
             (eq (qfirst (qcadadr x)) 'exit)
             (consp (qrest (qcadadr x)))
             (eq (qcddr (qcadadr x)) nil))
      collect
       (list (qcaadr x)
             (qsecond (qcadadr x)))))
    (setq before (take (|#| transform) z))
    (setq aft (|after| z before))
    (cond
     ((null before) (cons 'seq aft))
     ((null aft)
       (cons 'cond (append transform (list '(t (|conderr|))))))
     (t
       (cons 'cond (append transform
         (list (list ''t (|optSEQ| (cons 'seq aft))))))))))
  (getRidOfTemps (z)
   (let (g x r)
    (cond
     ((null z) nil)
     ((and (consp z) (consp (qfirst z)) (eq (qcaar z) 'let)
           (consp (qcdar z)) (consp (qcddar z))
           (gensymp (qcadar z))
           (> 2 (|numOfOccurencesOf| (qcadar z) (qrest z))))
       (setq g (qcadar z))
       (setq x (qcaddar z))
       (setq r (qrest z))
       (getRidOfTemps (subst x g r :test #'equal)))
     ((eq (car z) '|/throwAway|)
       (getRidOfTemps (cdr z)))
     (t
       (cons (car z) (getRidOfTemps (cdr z))))))))
 (tryToRemoveSEQ (SEQToCOND (getRidOfTemps (cdr arg))))))
```

———-

6.2.19 defplist optEQ

— postvars —

```
(eval-when (eval load)
```

```
 (setf (get 'eq 'optimize) '|optEQ|))
```

———-

6.2.20 defun optEQ

— defun optEQ —

```
(defun |optEQ| (u)
 (let (z r)
  (cond
   ((and (consp u) (eq (qfirst u) 'eq) (consp (qrest u))
         (consp (qcddr u)) (eq (qcdddr u) nil))
     (setq z (qsecond u))
     (setq r (qthird u))
; That undoes some weird work in Boolean to do with the definition of true
     (if (and (numberp z) (numberp r))
       (list 'quote (eq z r))
       u))
   (t u))))
```

———-

6.2.21 defplist optMINUS

— postvars —

```
(eval-when (eval load)
 (setf (get 'minus 'optimize) '|optMINUS|))
```

———-

6.2.22 defun optMINUS

— defun optMINUS —

```
(defun |optMINUS| (u)
 (let (v)
  (cond
    ((and (consp u) (eq (qfirst u) 'minus) (consp (qrest u))
          (eq (qcddr u) nil))
      (setq v (qsecond u))
      (cond ((numberp v) (- v)) (t u)))
    (t u))))
```

———-

6.2.23 defplist optQSMINUS

— postvars —

```
(eval-when (eval load)
 (setf (get 'qsminus 'optimize) '|optQSMINUS|))
```

———-

6.2.24 defun optQSMINUS

— defun optQSMINUS —

```
(defun |optQSMINUS| (u)
 (let (v)
  (cond
   ((and (consp u) (eq (qfirst u) 'qsminus) (consp (qrest u))
         (eq (qcddr u) nil))
     (setq v (qsecond u))
     (cond ((numberp v) (- v)) (t u)))
   (t u))))
```

———-

6.2.25 defplist opt-

— postvars —

```
(eval-when (eval load)
 (setf (get '- 'optimize) '|opt-|))
```

———-

6.2.26 defun opt-

— defun opt- —

```
(defun |opt-| (u)
 (let (v)
  (cond
   ((and (consp u) (eq (qfirst u) '-) (consp (qrest u))
         (eq (qcddr u) NIL))
    (setq v (qsecond u))
    (cond ((numberp v) (- v)) (t u)))
   (t u))))
```

———-

6.2.27 defplist optLESSP

— postvars —

```
(eval-when (eval load)
 (setf (get 'lessp 'optimize) '|optLESSP|))
```

———-

6.2.28 defun optLESSP

— defun optLESSP —

```
(defun |optLESSP| (u)
 (let (a b)
  (cond
   ((and (consp u) (eq (qfirst u) 'lessp) (consp (qrest u))
         (consp (qcddr u))
         (eq (qcdddr u) nil))
     (setq a (qsecond u))
     (setq b (qthird u))
     (if (eql b 0)
       (list 'minusp a)
       (list '> b a)))
   (t u))))
```

———-

6.2.29 defplist optSPADCALL

— postvars —

```
(eval-when (eval load)
 (setf (get 'spadcall 'optimize) '|optSPADCALL|))
```

———-

6.2.30 defun optSPADCALL

[optCall p217]
[$InteractiveMode p??]

— defun optSPADCALL —

```
(defun |optSPADCALL| (form)
 (let (fun argl tmp1 dom slot)
 (declare (special |$InteractiveMode|))
  (setq argl (cdr form))
```

```
  (cond
   ; last arg is function/env, but may be a form
   ((null |$InteractiveMode|) form)
   ((and (consp argl)
         (progn (setq tmp1 (reverse argl)) t)
         (consp tmp1))
     (setq fun (qfirst tmp1))
     (setq argl (qrest tmp1))
     (setq argl (nreverse argl))
     (cond
      ((and (consp fun)
            (eq (qfirst fun) 'elt)
            (progn
              (and (consp (qrest fun))
                   (progn
                    (setq dom (qsecond fun))
                    (and (consp (qcddr fun))
                         (eq (qcdddr fun) nil)
                         (progn
                           (setq slot (qthird fun))
                           t))))))
       (|optCall| (cons '|call| (cons (list 'elt dom slot) argl))))
      (t form)))
  (t form))))
```

———-

6.2.31 defplist optSuchthat

— postvars —

```
(eval-when (eval load)
 (setf (get '|\|| 'optimize) '|optSuchthat|))
```

———-

6.2.32 defun optSuchthat

— defun optSuchthat —

```
(defun |optSuchthat| (arg)
 (cons 'suchthat (cdr arg)))
```

———-

6.2.33 defplist optCatch

— postvars —

```
(eval-when (eval load)
 (setf (get 'catch 'optimize) '|optCatch|))
```

———-

6.2.34 defun optCatch

[rplac p??]
[optimize p213]
[$InteractiveMode p??]

— defun optCatch —

```
(defun |optCatch| (x)
 (labels (
  (changeThrowToExit (s g)
    (cond
     ((or (atom s) (member (car s) '(quote seq repeat collect))) nil)
     ((and (consp s) (eq (qfirst s) 'throw) (consp (qrest s))
           (equal (qsecond s) g))
        (|rplac| (car s) 'exit)
        (|rplac| (cdr s) (qcddr s)))
     (t
      (changeThrowToExit (car s) g)
      (changeThrowToExit (cdr s) g))))
  (hasNoThrows (a g)
    (cond
     ((and (consp a) (eq (qfirst a) 'throw) (consp (qrest a))
           (equal (qsecond a) g))
            nil)
     ((atom a) t)
     (t
      (and (hasNoThrows (car a) g)
           (hasNoThrows (cdr a) g)))))
  (changeThrowToGo (s g)
   (let (u)
    (cond
     ((or (atom s) (eq (car s) 'quote)) nil)
     ((and (consp s) (eq (qfirst s) 'throw) (consp (qrest s))
           (equal (qsecond s) g) (consp (qcddr s))
           (eq (qcdddr s) nil))
       (setq u (qthird s))
       (changeThrowToGo u g)
       (|rplac| (car s) 'progn)
       (|rplac| (cdr s) (list (list 'let (cadr g) u) (list 'go (cadr g)))))
     (t
      (changeThrowToGo (car s) g)
      (changeThrowToGo (cdr s) g))))))
 (let (g tmp2 u s tmp6 a)
 (declare (special |$InteractiveMode|))
   (setq g (cadr x))
   (setq a (caddr x))
   (cond
```

```
    (|$InteractiveMode| x)
    ((atom a) a)
    (t
     (cond
      ((and (consp a) (eq (qfirst a) 'seq) (consp (qrest a))
            (progn (setq tmp2 (reverse (qrest a))) t)
            (consp tmp2) (consp (qfirst tmp2)) (eq (qcaar tmp2) 'throw)
            (consp (qcdar tmp2))
            (equal (qcadar tmp2) g)
            (consp (qcddar tmp2))
            (eq (qcdddar tmp2) nil))
      (setq u (qcaddar tmp2))
      (setq s (qrest tmp2))
      (setq s (nreverse s))
      (changeThrowToExit s g)
      (|rplac| (cdr a) (append s (list (list 'exit u))))
      (setq tmp6 (|optimize| x))
      (setq a (caddr tmp6))))
     (cond
      ((hasNoThrows a g)
        (|rplac| (car x) (car a))
        (|rplac| (cdr x) (cdr a)))
      (t
        (changeThrowToGo a g)
        (|rplac| (car x) 'seq)
        (|rplac| (cdr x)
          (list (list 'exit a) (cadr g) (list 'exit (cadr g))))))
     x)))))
```

———-

6.2.35 defplist optCond

— postvars —

```
(eval-when (eval load)
 (setf (get 'cond 'optimize) '|optCond|))
```

———-

6.2.36 defun optCond

[rplacd p??]
[TruthP p248]
[EqualBarGensym p230]
[rplac p??]

— defun optCond —

```
(defun |optCond| (x)
 (let (z p1 p2 c3 c1 c2 a result)
```

```
(setq z (cdr x))
(when
 (and (consp z) (consp (qrest z)) (eq (qcddr z) nil)
      (consp (qsecond z)) (consp (qcdadr z))
      (eq (qrest (qcdadr z)) nil)
      (|TruthP| (qcaadr z))
      (consp (qcadadr z))
      (eq (qfirst (qcadadr z)) 'cond))
  (rplacd (cdr x) (qrest (qcadadr z))))
 (cond
  ((and (consp z) (consp (qfirst z)) (consp (qrest z)) (consp (qsecond z)))
    (setq p1 (qcaar z))
    (setq c1 (qcdar z))
    (setq p2 (qcaadr z))
    (setq c2 (qcdadr z))
    (when
      (or (and (consp p1) (eq (qfirst p1) 'null) (consp (qrest p1))
               (eq (qcddr p1) nil)
               (equal (qsecond p1) p2))
          (and (consp p2) (eq (qfirst p2) 'null) (consp (qrest p2))
               (eq (qcddr p2) nil)
               (equal (qsecond p2) p1)))
       (setq z (list (cons p1 c1) (cons ''t c2)))
       (rplacd x z))
    (when
     (and (consp c1) (eq (qrest c1) nil) (equal (qfirst c1) 'nil)
          (equal p2 ''t) (equal (car c2) ''t))
      (if (and (consp p1) (eq (qfirst p1) 'null) (consp (qrest p1))
               (eq (qcddr p1) nil))
          (setq result (qsecond p1))
          (setq result (list 'null p1))))))
(if result
 result
 (cond
  ((and (consp z) (consp (qfirst z)) (consp (qrest z)) (consp (qsecond z))
        (consp (qcddr z)) (eq (qcdddr z) nil)
        (consp (qthird z))
        (|TruthP| (qcaaddr z)))
    (setq p1 (qcaar z))
    (setq c1 (qcdar z))
    (setq p2 (qcaadr z))
    (setq c2 (qcdadr z))
    (setq c3 (qcdaddr z))
    (cond
     ((|EqualBarGensym| c1 c3)
      (list 'cond
       (cons (list 'or p1 (list 'null p2)) c1) (cons (list 'quote t) c2)))
     ((|EqualBarGensym| c1 c2)
      (list 'cond (cons (list 'or p1 p2) c1) (cons (list 'quote t) c3)))
     (t x)))
  (t
   (do ((y z (cdr y)))
       ((atom y) nil)
     (do ()
```

```
          ((null (and (consp y) (consp (qfirst y)) (consp (qcdar y))
                      (eq (qcddar y) nil) (consp (qrest y))
                      (consp (qsecond y)) (consp (qcdadr y))
                      (eq (qcddadr y) nil)
                      (|EqualBarGensym| (qcadar y)
                                        (qcadadr y))))
            nil)
        (setq a (list 'or (qcaar y) (qcaadr y)))
        (rplac (car (car y)) a)
        (rplac (cdr y) (qcddr y))))
     x)))))
```

———-

6.2.37 defun EqualBarGensym

[gensymp p??]
[$GensymAssoc p??]
[$GensymAssoc p??]

— defun EqualBarGensym —

```
(defun |EqualBarGensym| (x y)
 (labels (
  (fn (x y)
   (let (z)
   (declare (special |$GensymAssoc|))
    (cond
     ((equal x y) t)
     ((and (gensymp x) (gensymp y))
      (if (setq z (|assoc| x |$GensymAssoc|))
        (if (equal y (cdr z)) t nil)
        (progn
         (setq |$GensymAssoc| (cons (cons x y) |$GensymAssoc|))
         t)))
     ((null x) (and (consp y) (eq (qrest y) nil) (gensymp (qfirst y))))
     ((null y) (and (consp x) (eq (qrest x) nil) (gensymp (qfirst x))))
     ((or (atom x) (atom y)) nil)
     (t
      (and (fn (car x) (car y))
           (fn (cdr x) (cdr y))))))))
 (let (|$GensymAssoc|)
 (declare (special |$GensymAssoc|))
  (setq |$GensymAssoc| NIL)
  (fn x y))))
```

———-

6.2.38 defplist optMkRecord

— postvars —

```
(eval-when (eval load)
 (setf (get '|mkRecord| 'optimize) '|optMkRecord|))
```

———-

6.2.39 defun optMkRecord

[length p??]

— defun optMkRecord —

```
(defun |optMkRecord| (arg)
 (let (u)
  (setq u (cdr arg))
  (cond
   ((and (consp u) (eq (qrest u) nil)) (list 'list (qfirst u)))
   ((eql (|#| u) 2) (cons 'cons u))
   (t (cons 'vector u)))))
```

———-

6.2.40 defplist optRECORDELT

— postvars —

```
(eval-when (eval load)
 (setf (get 'recordelt 'optimize) '|optRECORDELT|))
```

———-

6.2.41 defun optRECORDELT

[keyedSystemError p??]

— defun optRECORDELT —

```
(defun |optRECORDELT| (arg)
 (let (name ind len)
  (setq name (cadr arg))
  (setq ind (caddr arg))
  (setq len (cadddr arg))
  (cond
   ((eql len 1)
    (cond
     ((eql ind 0) (list 'qcar name))
```

```
    (t (|keyedSystemError| "Bad index in record optimization: %1"
         (list ind)))))
  ((eql len 2)
   (cond
    ((eql ind 0) (list 'qcar name))
    ((eql ind 1) (list 'qcdr name))
    (t (|keyedSystemError| "Bad index in record optimization: %1"
        (list ind)))))
  (t (list 'qvelt name ind)))))
```

———-

6.2.42 defplist optSETRECORDELT

— postvars —

```
(eval-when (eval load)
 (setf (get 'setrecordelt 'optimize) '|optSETRECORDELT|))
```

———-

6.2.43 defun optSETRECORDELT

[keyedSystemError p??]

— defun optSETRECORDELT —

```
(defun |optSETRECORDELT| (arg)
 (let (name ind len expr)
  (setq name (cadr arg))
  (setq ind (caddr arg))
  (setq len (cadddr arg))
  (setq expr (car (cddddr arg)))
  (cond
   ((eql len 1)
    (if (eql ind 0)
      (list 'progn (list 'rplaca name expr) (list 'qcar name))
      (|keyedSystemError| "Bad index in record optimization: %1" (list ind))))
   ((eql len 2)
    (cond
     ((eql ind 0)
       (list 'progn (list 'rplaca name expr) (list 'qcar name)))
     ((eql ind 1)
       (list 'progn (list 'rplacd name expr) (list 'qcdr name)))
     (t (|keyedSystemError| "Bad index in record optimization: %1"
         (list ind)))))
   (t
     (list 'qsetvelt name ind expr)))))
```

———-

6.2.44 defplist optRECORDCOPY

— postvars —

```
(eval-when (eval load)
 (setf (get 'recordcopy 'optimize) '|optRECORDCOPY|))
```

———-

6.2.45 defun optRECORDCOPY

— defun optRECORDCOPY —

```
(defun |optRECORDCOPY| (arg)
 (let (name len)
  (setq name (cadr arg))
  (setq len (caddr arg))
  (cond
   ((eql len 1) (list 'list (list 'car name)))
   ((eql len 2) (list 'cons (list 'car name) (list 'cdr name)))
   (t           (list 'replace (list 'make-array len) name)))))
```

———-

6.3 Functions to manipulate modemaps

6.3.1 defun addDomain

[identp p??]
[qslessp p??]
[getDomainsInScope p235]
[domainMember p244]
[isLiteral p??]
[addNewDomain p236]
[getmode p??]
[isCategoryForm p??]
[isFunctor p234]
[constructor? p??]
[member p??]
[unknownTypeError p234]

— defun addDomain —

```
(defun |addDomain| (domain env)
 (let (s name tmp1)
  (cond
   ((atom domain)
     (cond
```

```
      ((eq domain '|$EmptyMode|) env)
      ((eq domain '|$NoValueMode|) env)
      ((or (null (identp domain))
           (and (qslessp 2 (|#| (setq s (princ-to-string domain))))
                (eq #\# (elt s 0))
                (eq #\# (elt s 1))))
            env)
      ((member domain (|getDomainsInScope| env)) env)
      ((|isLiteral| domain env) env)
      (t (|addNewDomain| domain env))))
   ((eq (setq name (car domain)) '|Category|) env)
   ((|domainMember| domain (|getDomainsInScope| env)) env)
   ((and (progn
          (setq tmp1 (|getmode| name env))
          (and (consp tmp1) (eq (qfirst tmp1) '|Mapping|)
               (consp (qrest tmp1))))
          (|isCategoryForm| (second tmp1) env))
     (|addNewDomain| domain env))
   ((or (|isFunctor| name) (|constructor?| name))
     (|addNewDomain| domain env))
   (t
     (when (and (null (|isCategoryForm| domain env))
                (null (|member| name '(|Mapping| category))))
       (|unknownTypeError| name))
     env))))
```

———-

6.3.2 defun unknownTypeError

[stackSemanticError p??]

— defun unknownTypeError —

```
(defun |unknownTypeError| (name)
 (let (op)
  (setq name
   (if (and (consp name) (setq op (qfirst name)))
    op
    name))
  (|stackSemanticError| (list name '|is not a known type|) nil)))
```

———-

6.3.3 defun isFunctor

[opOf p??]
[identp p??]
[getdatabase p??]
[get p??]
[constructor? p??]

[updateCategoryFrameForCategory p116]
[updateCategoryFrameForConstructor p115]
[$CategoryFrame p??]
[$InteractiveMode p??]

— defun isFunctor —

```
(defun |isFunctor| (x)
 (let (op u prop)
 (declare (special |$CategoryFrame| |$InteractiveMode|))
  (setq op (|opOf| x))
  (cond
   ((null (identp op)) nil)
   (|$InteractiveMode|
    (if (member op '(|Union| |SubDomain| |Mapping| |Record|))
     t
     (member (getdatabase op 'constructorkind) '(|domain| |package|))))
   ((setq u
     (or (|get| op '|isFunctor| |$CategoryFrame|)
         (member op '(|SubDomain| |Union| |Record|))))
      u)
   ((|constructor?| op)
     (cond
      ((setq prop (|get| op '|isFunctor| |$CategoryFrame|)) prop)
      (t
       (if (eq (getdatabase op 'constructorkind) '|category|)
         (|updateCategoryFrameForCategory| op)
         (|updateCategoryFrameForConstructor| op))
       (|get| op '|isFunctor| |$CategoryFrame|))))
   (t nil))))
```

———-

6.3.4 defun getDomainsInScope

The way XLAMs work:

```
((XLAM ($1 $2 $3) (SETELT $1 0 $3)) X "c" V) ==> (SETELT X 0 V)
```

[get p??]
[$CapsuleDomainsInScope p??]
[$insideCapsuleFunctionIfTrue p??]

— defun getDomainsInScope —

```
(defun |getDomainsInScope| (env)
  (declare (special |$CapsuleDomainsInScope| |$insideCapsuleFunctionIfTrue|))
  (if |$insideCapsuleFunctionIfTrue|
     |$CapsuleDomainsInScope|
     (|get| '|$DomainsInScope| 'special env)))
```

———-

6.3.5 defun putDomainsInScope

[getDomainsInScope p235]
[put p??]
[delete p??]
[say p??]
[member p??]
[$CapsuleDomainsInScope p??]
[$insideCapsuleFunctionIfTrue p??]

— defun putDomainsInScope —

```
(defun |putDomainsInScope| (x env)
 (let (z newValue)
 (declare (special |$CapsuleDomainsInScope| |$insideCapsuleFunctionIfTrue|))
  (setq z (|getDomainsInScope| env))
  (when (|member| x z) (say "****** Domain: " x " already in scope"))
  (setq newValue (cons x (|delete| x z)))
  (if |$insideCapsuleFunctionIfTrue|
    (progn
      (setq |$CapsuleDomainsInScope| newValue)
      env)
    (|put| '|$DomainsInScope| 'special newValue env))))
```

———-

6.3.6 defun isSuperDomain

[isSubset p??]
[lassoc p??]
[opOf p??]
[get p??]

— defun isSuperDomain —

```
(defun |isSuperDomain| (domainForm domainFormp env)
 (cond
   ((|isSubset| domainFormp domainForm env) t)
   ((and (eq domainForm '|Rep|) (eq domainFormp '$)) t)
   (t (lassoc (|opOf| domainFormp) (|get| domainForm '|SubDomain| env)))))
```

———-

6.3.7 defun addNewDomain

[augModemapsFromDomain p237]

— defun addNewDomain —

```
(defun |addNewDomain| (domain env)
  (|augModemapsFromDomain| domain domain env))
```

———-

6.3.8 defun augModemapsFromDomain

[member p??]
[getDomainsInScope p235]
[getdatabase p??]
[opOf p??]
[addNewDomain p236]
[listOrVectorElementNode p??]
[stripUnionTags p??]
[augModemapsFromDomain1 p237]
[$Category p??]
[$DummyFunctorNames p??]

— defun augModemapsFromDomain —

```
(defun |augModemapsFromDomain| (name functorForm env)
 (let (curDomainsInScope u innerDom)
 (declare (special |$Category| |$DummyFunctorNames|))
  (cond
   ((|member| (or (ifcar name) name) |$DummyFunctorNames|)
     env)
   ((or (equal name |$Category|) (|isCategoryForm| name env))
     env)
   ((|member| name (setq curDomainsInScope  (|getDomainsInScope| env)))
     env)
   (t
    (when (setq u (getdatabase (|opOf| functorForm) 'superdomain))
      (setq env (|addNewDomain| (car u) env)))
    (when (setq innerDom (|listOrVectorElementMode| name))
      (setq env (|addDomain| innerDom env)))
    (when (and (consp name) (eq (qfirst name) '|Union|))
      (dolist (d (|stripUnionTags| (qrest name)))
        (setq env (|addDomain| d env))))
    (|augModemapsFromDomain1| name functorForm env)))))
```

———-

6.3.9 defun augModemapsFromDomain1

[getl p??]
[addConstructorModemaps p238]
[getmode p??]
[augModemapsFromCategory p244]
[getmodeOrMapping p??]
[substituteCategoryArguments p238]
[stackMessage p??]

— defun augModemapsFromDomain1 —

```
(defun |augModemapsFromDomain1| (name functorForm env)
 (let (mappingForm categoryForm functArgTypes catform)
  (cond
   ((getl (ifcar functorForm) '|makeFunctionList|)
     (|addConstructorModemaps| name functorForm env))
   ((and (atom functorForm) (setq catform (|getmode| functorForm env)))
     (|augModemapsFromCategory| name functorForm catform env))
   ((setq mappingForm (|getmodeOrMapping| (ifcar functorForm) env))
     (when (eq (car mappingForm) '|Mapping|) (car mappingForm))
     (setq categoryForm (cadr mappingForm))
     (setq functArgTypes (cddr mappingForm))
     (setq catform
       (|substituteCategoryArguments| (cdr functorForm) categoryForm))
     (|augModemapsFromCategory| name functorForm catform env))
   (t
     (|stackMessage| (list functorForm '| is an unknown mode|))
     env))))
```

———-

6.3.10 defun substituteCategoryArguments

[internl p??]
[sublis p??]

— defun substituteCategoryArguments —

```
(defun |substituteCategoryArguments| (argl catform)
 (let (arglAssoc (i 0))
  (setq argl (subst '$$ '$ argl :test #'equal))
  (setq arglAssoc
   (loop for a in argl
    collect (cons (internl '|#| (princ-to-string (incf i))) a)))
  (sublis arglAssoc catform)))
```

———-

6.3.11 defun addConstructorModemaps

[putDomainsInScope p236]
[getl p??]
[addModemap p252]
[$InteractiveMode p??]

— defun addConstructorModemaps —

```
(defun |addConstructorModemaps| (name form env)
 (let (|$InteractiveMode| functorName fn tmp1 funList op sig nsig opcode)
 (declare (special |$InteractiveMode|))
  (setq functorName (car form))
```

```
  (setq |$InteractiveMode| nil)
  (setq env (|putDomainsInScope| name env))
  (setq fn (getl functorName '|makeFunctionList|))
  (setq tmp1 (funcall fn name form env))
  (setq funList (car tmp1))
  (setq env (cadr tmp1))
  (dolist (item funList)
    (setq op (first item))
    (setq sig (second item))
    (setq opcode (third item))
    (when (and (consp opcode) (consp (qrest opcode))
               (consp (qcddr opcode))
               (eq (qcdddr opcode) nil)
               (eq (qfirst opcode) 'elt))
       (setq nsig (subst '$$$ name sig :test #'equal))
       (setq nsig
        (subst '$ '$$$ (subst '$$ '$ nsig :test #'equal) :test #'equal))
       (setq opcode (list (first opcode) (second opcode) nsig)))
    (setq env (|addModemap| op name sig t opcode env)))
  env))
```

———-

6.3.12 defun getModemap

[get p??]
[compApplyModemap p239]
[sublis p??]

— defun getModemap —

```
(defun |getModemap| (x env)
 (let (u)
  (dolist (modemap (|get| (first x) '|modemap| env))
   (when (setq u (|compApplyModemap| x modemap env nil))
     (return (sublis (third u) modemap))))))
```

———-

6.3.13 defun compApplyModemap

[length p??]
[pmatchWithSl p??]
[sublis p??]
[comp p530]
[coerce p325]
[compMapCond p241]
[member p??]
[genDeltaEntry p??]
[$e p??]
[$bindings p??]

[$e p??]
[$bindings p??]

— defun compApplyModemap —

```
(defun |compApplyModemap| (form modemap |$e| sl)
 (declare (special |$e|))
 (let (op argl mc mr margl fnsel g mp lt ltp temp1 f)
 (declare (special |$bindings| |$e|))
  ;  -- $e     is the current environment
  ;  -- sl     substitution list, nil means bottom-up, otherwise top-down
  ;  -- 0.  fail immediately if #argl=#margl
  (setq op (car form))
  (setq argl (cdr form))
  (setq mc (caar modemap))
  (setq mr (cadar modemap))
  (setq margl (cddar modemap))
  (setq fnsel (cdr modemap))
  (when (= (|#| argl) (|#| margl))
   ; 1.  use modemap to evaluate arguments, returning failed if not possible
   (setq lt
    (prog (t0)
     (return
      (do ((t1 argl (cdr t1)) (y NIL) (t2 margl (cdr t2)) (m nil))
          ((or (atom t1) (atom t2)) (nreverse0 t0))
        (setq y (car t1))
        (setq m (car t2))
        (setq t0
         (cons
          (progn
           (setq sl (|pmatchWithSl| mp m sl))
           (setq g (sublis sl m))
           (setq temp1 (or (|comp| y g |$e|) (return '|failed|)))
           (setq mp (cadr temp1))
           (setq |$e| (caddr temp1))
           temp1)
            t0)))))))
   ; 2.  coerce each argument to final domain, returning failed
   ;     if not possible
   (unless (eq lt '|failed|)
     (setq ltp
      (loop for y in lt for d in (sublis sl margl)
       collect (or (|coerce| y d) (return '|failed|))))
     (unless (eq ltp '|failed|)
       ; 3.  obtain domain-specific function, if possible, and return
       ; $bindings is bound by compMapCond
       (setq temp1 (|compMapCond| op mc sl fnsel))
       (when temp1
        ; can no longer trust what the modemap says for a reference into
        ; an exterior domain (it is calculating the displacement based on view
        ; information which is no longer valid; thus ignore this index and
        ; store the signature instead.)
        (setq f (car temp1))
        (setq |$bindings| (cadr temp1))
```

```
      (if (and (consp f) (consp (qcdr f)) (consp (qcddr f)) ; f is [op1,.]
               (eq (qcdddr f) nil)
               (|member| (qcar f) '(elt const |Subsumed|)))
        (list (|genDeltaEntry| (cons op modemap)) ltp |$bindings|)
        (list f ltp |$bindings|)))))))
```

———-

6.3.14 defun compMapCond

[compMapCond' p241]
[$bindings p??]

— defun compMapCond —

```
(defun |compMapCond| (op mc |$bindings| fnsel)
 (declare (special |$bindings|))
 (let (t0)
  (do ((t1 nil t0) (t2 fnsel (cdr t2)) (u nil))
      ((or t1 (atom t2) (progn (setq u (car t2)) nil)) t0)
   (setq t0 (or t0 (|compMapCond'| u op mc |$bindings|))))))
```

———-

6.3.15 defun compMapCond'

[compMapCond'' p241]
[compMapConfFun p??]
[stackMessage p??]

— defun compMapCond' —

```
(defun |compMapCond'| (t0 op dc bindings)
 (let ((cexpr (car t0)) (fnexpr (cadr t0)))
  (if (|compMapCond''| cexpr dc)
    (|compMapCondFun| fnexpr op dc bindings)
    (|stackMessage| `("not known that" ,dc "has" ,cexpr)))))
```

———-

6.3.16 defun compMapCond''

[compMapCond'' p241]
[knownInfo p??]
[get p??]
[stackMessage p??]
[$Information p??]
[$e p??]

— defun compMapCond'' —

```
(defun |compMapCond''| (cexpr dc)
 (let (l u tmp1 tmp2)
 (declare (special |$Information| |$e|))
  (cond
   ((eq cexpr t) t)
   ((and (consp cexpr)
         (eq (qcar cexpr) 'and)
         (progn (setq l (qcdr cexpr)) t))
     (prog (t0)
      (setq t0 t)
      (return
       (do ((t1 nil (null t0)) (t2 l (cdr t2)) (u nil))
           ((or t1 (atom t2) (progn (setq u (car t2)) nil)) t0)
        (setq t0 (and t0 (|compMapCond''| u dc)))))))
   ((and (consp cexpr)
         (eq (qcar cexpr) 'or)
         (progn (setq l (qcdr cexpr)) t))
    (prog (t3)
     (setq t3 nil)
     (return
      (do ((t4 nil t3) (t5 l (cdr t5)) (u nil))
          ((or t4 (atom t5) (progn (setq u (car t5)) nil)) t3)
         (setq t3 (or t3 (|compMapCond''| u dc)))))))
   ((and (consp cexpr)
         (eq (qcar cexpr) '|not|)
         (progn
          (setq tmp1 (qcdr cexpr))
          (and (consp tmp1)
               (eq (qcdr tmp1) nil)
               (progn (setq u (qcar tmp1)) t))))
     (null (|compMapCond''| u dc)))
   ((and (consp cexpr)
         (eq (qcar cexpr) '|has|)
         (progn
          (setq tmp1 (qcdr cexpr))
          (and (consp tmp1)
               (progn
                (setq tmp2 (qcdr tmp1))
                (and (consp tmp2)
                     (eq (qcdr tmp2) nil))))))
     (cond
      ((|knownInfo| cexpr) t)
      (t nil)))
   ((|member|
      (cons 'attribute (cons dc (cons cexpr nil)))
      (|get| '|$Information| 'special |$e|))
     t)
   (t
    (|stackMessage| `("not known that" ,dc "has" ,cexpr))
    nil))))
```

———

6.3.17 defun compMapCondFun

— defun compMapCondFun —

```
(defun |compMapCondFun| (fnexpr op dc bindings)
 (declare (ignore op) (ignore dc))
 (cons fnexpr (cons bindings nil)))
```

———-

6.3.18 defun getUniqueSignature

[getUniqueModemap p243]

— defun getUniqueSignature —

```
(defun |getUniqueSignature| (form env)
  (cdar (|getUniqueModemap| (first form) (|#| (rest form)) env)))
```

———-

6.3.19 defun getUniqueModemap

[getModemapList p243]
[qslessp p??]
[stackWarning p??]

— defun getUniqueModemap —

```
(defun |getUniqueModemap| (op numOfArgs env)
 (let (mml)
  (cond
   ((eql 1 (|#| (setq mml (|getModemapList| op numOfArgs env))))
     (car mml))
   ((qslessp 1 (|#| mml))
     (|stackWarning|
       (list numOfArgs " argument form of: " op " has more than one modemap"))
     (car mml))
   (t nil))))
```

———-

6.3.20 defun getModemapList

[getModemapListFromDomain p244]
[nreverse0 p??]
[get p??]

— defun getModemapList —

```
(defun |getModemapList| (op numOfArgs env)
 (let (result)
  (cond
   ((and (consp op) (eq (qfirst op) '|elt|) (consp (qrest op))
         (consp (qcddr op)) (eq (qcdddr op) nil))
     (|getModemapListFromDomain| (third op) numOfArgs (second op) env))
  (t
   (dolist (term (|get| op '|modemap| env) (nreverse0 result))
     (when (eql numOfArgs (|#| (cddar term))) (push term result)))))))
```

———-

6.3.21 defun getModemapListFromDomain

[get p??]

— defun getModemapListFromDomain —

```
(defun |getModemapListFromDomain| (op numOfArgs d env)
 (loop for term in (|get| op '|modemap| env)
       when (and (equal (caar term) d) (eql (|#| (cddar term)) numOfArgs))
       collect term))
```

———-

6.3.22 defun domainMember

[modeEqual p335]

— defun domainMember —

```
(defun |domainMember| (dom domList)
 (let (result)
  (dolist (d domList result)
   (setq result (or result (|modeEqual| dom d))))))
```

———-

6.3.23 defun augModemapsFromCategory

[evalAndSub p248]
[compilerMessage p??]
[putDomainsInScope p236]
[addModemapKnown p251]
[$base p??]

— defun augModemapsFromCategory —

```
(defun |augModemapsFromCategory| (domainName functorform categoryForm env)
 (let (tmp1 op sig cond fnsel)
 (declare (special |$base|))
```

```
  (setq tmp1 (|evalAndSub| domainName domainName functorform categoryForm env))
  (|compilerMessage| (list '|Adding | domainName '| modemaps|))
  (setq env (|putDomainsInScope| domainName (second tmp1)))
  (setq |$base| 4)
  (dolist (u (first tmp1))
    (setq op (caar u))
    (setq sig (cadar u))
    (setq cond (cadr u))
    (setq fnsel (caddr u))
    (setq env (|addModemapKnown| op domainName sig cond fnsel env)))
  env))
```

———-

6.3.24 defun addEltModemap

This is a hack to change selectors from strings to identifiers; and to add flag identifiers as literals in the environment [makeLiteral p??]
[addModemap1 p253]
[systemErrorHere p??]
[$insideCapsuleFunctionIfTrue p??]
[$e p??]

— defun addEltModemap —

```
(defun |addEltModemap| (op mc sig pred fn env)
 (let (tmp1 v sel lt id)
 (declare (special |$e| |$insideCapsuleFunctionIfTrue|))
  (cond
   ((and (eq op '|elt|) (consp sig))
     (setq tmp1 (reverse sig))
     (setq sel (qfirst tmp1))
     (setq lt (nreverse (qrest tmp1)))
     (cond
       ((stringp sel)
         (setq id (intern sel))
         (if |$insideCapsuleFunctionIfTrue|
           (setq |$e| (|makeLiteral| id |$e|))
           (setq env (|makeLiteral| id env)))
         (|addModemap1| op mc (append lt (list id)) pred fn env))
       (t (|addModemap1| op mc sig pred fn env))))
   ((and (eq op '|setelt|) (consp sig))
     (setq tmp1 (reverse sig))
     (setq v (qfirst tmp1))
     (setq sel (qsecond tmp1))
     (setq lt (nreverse (qcddr tmp1)))
     (cond
       ((stringp sel) (setq id (intern sel))
         (if |$insideCapsuleFunctionIfTrue|
           (setq |$e| (|makeLiteral| id |$e|))
           (setq env (|makeLiteral| id env)))
         (|addModemap1| op mc (append lt (list id v)) pred fn env))
```

```
      (t (|addModemap1| op mc sig pred fn env))))
   (t (|systemErrorHere| "addEltModemap")))))
```

———-

6.3.25 defun mkNewModemapList

[member p??]
[assoc p??]
[mergeModemap p247]
[nreverse0 p??]
[insertModemap p247]
[$InteractiveMode p??]
[$forceAdd p??]

— **defun mkNewModemapList** —

```
(defun |mkNewModemapList| (mc sig pred fn curModemapList env filenameOrNil)
 (let (map entry oldMap opred result)
 (declare (special |$InteractiveMode| |$forceAdd|))
   (setq entry
    (cons (setq map (cons mc sig)) (cons (list pred fn) filenameOrNil)))
   (cond
    ((|member| entry curModemapList) curModemapList)
    ((and (setq oldMap (|assoc| map curModemapList))
          (consp oldMap) (consp (qrest oldMap))
          (consp (qsecond oldMap))
          (consp (qcdadr oldMap))
          (eq (qcddadr oldMap) nil)
          (equal (qcadadr oldMap) fn))
      (setq opred (qcaadr oldMap))
      (cond
       (|$forceAdd| (|mergeModemap| entry curModemapList env))
       ((eq opred t) curModemapList)
       (t
         (when (and (not (eq pred t)) (not (equal pred opred)))
            (setq pred (list 'or pred opred)))
         (dolist (x curModemapList (nreverse0 result))
          (push
           (if (equal x oldMap)
             (cons map (cons (list pred fn) filenameOrNil))
             x)
           result)))))
    (|$InteractiveMode|
     (|insertModemap| entry curModemapList))
    (t
     (|mergeModemap| entry curModemapList env)))))
```

———-

6.3.26 defun insertModemap

— defun insertModemap —

```
(defun |insertModemap| (new mmList)
 (if (null mmList) (list new) (cons new mmList)))
```

———-

6.3.27 defun mergeModemap

[isSuperDomain p236]
[TruthP p248]
[$forceAdd p??]

— defun mergeModemap —

```
(defun |mergeModemap| (entry modemapList env)
 (let (mc sig pred mcp sigp predp newmm mm)
 (declare (special |$forceAdd|))
  ; break out the condition, signature, and predicate fields of the new entry
  (setq mc (caar entry))
  (setq sig (cdar entry))
  (setq pred (caadr entry))
  (seq
   ; walk across the successive tails of the modemap list
   (do ((mmtail modemapList (cdr mmtail)))
       ((atom mmtail) nil)
     (setq mcp (caaar mmtail))
     (setq sigp (cdaar mmtail))
     (setq predp (caadar mmtail))
     (cond
      ((or (equal mc mcp) (|isSuperDomain| mcp mc env))
        ; if this is a duplicate condition
        (exit
         (progn
          (setq newmm nil)
          (setq mm modemapList)
          ; copy the unique modemap terms
          (loop while (not (eq mm mmtail)) do
            (setq newmm (cons (car mm) newmm))
            (setq mm (cdr mm)))
          ; if the conditions and signatures are equal
          (when (and (equal mc mcp) (equal sig sigp))
            ; we only need one of these unless the conditions are hairy
            (cond
             ((and (null |$forceAdd|) (|TruthP| predp))
               ; the new predicate buys us nothing
               (setq entry nil)
               (return modemapList))
             ((|TruthP| pred)
               ; the thing we matched against is useless, by comparison
```

```
               (setq mmtail (cdr mmtail)))))
          (setq modemapList (nconc (nreverse newmm) (cons entry mmtail)))
          (setq entry nil)
          (return modemapList))))))
   ; if the entry is still defined, add it to the modemap
   (if entry
     (append modemapList (list entry))
     modemapList))))
```

———-

6.3.28 defun TruthP

— defun TruthP —

```
(defun |TruthP| (x)
 (cond
   ((null x) nil)
   ((eq x t) t)
   ((and (consp x) (eq (qfirst x) 'quote)) t)
   (t nil)))
```

———-

6.3.29 defun evalAndSub

[isCategory p??]
[substNames p250]
[contained p??]
[put p??]
[get p??]
[getOperationAlist p249]
[$lhsOfColon p??]

— defun evalAndSub —

```
(defun |evalAndSub| (domainName viewName functorForm form |$e|)
 (declare (special |$e|))
 (let (|$lhsOfColon| opAlist substAlist)
  (declare (special |$lhsOfColon|))
   (setq |$lhsOfColon| domainName)
   (cond
    ((|isCategory| form)
      (list (|substNames| domainName viewName functorForm (elt form 1)) |$e|))
    (t
     (when (contained '$$ form)
       (setq |$e| (|put| '$$ '|mode| (|get| '$ '|mode| |$e|) |$e|)))
     (setq opAlist (|getOperationAlist| domainName functorForm form))
     (setq substAlist (|substNames| domainName viewName functorForm opAlist))
     (list substAlist |$e|)))))
```

———-

6.3.30 defun getOperationAlist

[getdatabase p??]
[isFunctor p234]
[systemError p??]
[compMakeCategoryObject p198]
[stackMessage p??]
[$e p??]
[$domainShell p??]
[$insideFunctorIfTrue p??]
[$functorForm p??]

— defun getOperationAlist —

```
(defun |getOperationAlist| (name functorForm form)
 (let (u tt)
 (declare (special |$e| |$domainShell| |$insideFunctorIfTrue| |$functorForm|))
  (when (and (atom name) (getdatabase name 'niladic))
    (setq functorform (list functorForm)))
  (cond
   ((and (setq u (|isFunctor| functorForm))
         (null (and |$insideFunctorIfTrue|
                    (equal (first functorForm) (first |$functorForm|)))))
    u)
   ((and |$insideFunctorIfTrue| (eq name '$))
    (if |$domainShell|
     (elt |$domainShell| 1)
     (|systemError| "$ has no shell now")))
   ((setq tt (|compMakeCategoryObject| form |$e|))
    (setq |$e| (third tt))
    (elt (first tt) 1))
   (t
    (|stackMessage| (list '|not a category form: | form))))))
```

———-

6.3.31 defvar $FormalMapVariableList

— initvars —

```
(defvar |$FormalMapVariableList|
  '(\#1 \#2 \#3 \#4 \#5 \#6 \#7 \#8 \#9 \#10 \#11 \#12 \#13 \#14 \#15))
```

———-

6.3.32 defun substNames

[isCategoryPackageName p199]
[eqsubstlist p??]
[nreverse0 p??]
[$FormalMapVariableList p249]

— defun substNames —

```
(defun |substNames| (domainName viewName functorForm opalist)
 (let (nameForDollar sel pos modemapform tmp0 tmp1)
 (declare (special |$FormalMapVariableList|))
  (setq functorForm (subst '$$ '$ functorForm))
  (setq nameForDollar
   (if (|isCategoryPackageName| functorForm)
     (second functorForm)
     domainName))
; following calls to SUBSTQ must copy to save RPLAC's in
; putInLocalDomainReferences
  (dolist (term
            (eqsubstlist (ifcdr functorForm) |$FormalMapVariableList| opalist)
            (nreverse0 tmp0))
   (setq tmp1 (reverse term))
   (setq sel (caar tmp1))
   (setq pos (caddar tmp1))
   (setq modemapform (nreverse (cdr tmp1)))
  (push
    (append
     (subst '$ '$$ (subst nameForDollar '$ modemapform))
     (list
       (list sel viewName (if (eq domainName '$) pos (cadar modemapform)))))
    tmp0))))
```

———-

6.3.33 defun augModemapsFromCategoryRep

[evalAndSub p248]
[isCategory p??]
[compilerMessage p??]
[putDomainsInScope p236]
[assoc p??]
[addModemap p252]
[$base p??]

— defun augModemapsFromCategoryRep —

```
(defun |augModemapsFromCategoryRep|
         (domainName repDefn functorBody categoryForm env)
 (labels (
  (redefinedList (op z)
   (let (result)
```

```
    (dolist (u z result)
     (setq result (or result (redefined op u))))))
  (redefined (opname u)
   (let (op z result)
   (when (consp u)
    (setq op (qfirst u))
    (setq z (qrest u))
    (cond
     ((eq op 'def) (equal opname (caar z)))
     ((member op '(progn seq)) (redefinedList opname z))
     ((eq op 'cond)
       (dolist (v z result)
         (setq result (or result (redefinedList opname (cdr v))))))))))
 (let (fnAlist tmp1 repFnAlist catform lhs op sig cond fnsel u)
 (declare (special |$base|))
  (setq tmp1 (|evalAndSub| domainName domainName domainName categoryForm env))
  (setq fnAlist (car tmp1))
  (setq env (cadr tmp1))
  (setq tmp1 (|evalAndSub| '|Rep| '|Rep| repDefn (|getmode| repDefn env) env))
  (setq repFnAlist (car tmp1))
  (setq env (cadr tmp1))
  (setq catform
    (if (|isCategory| categoryForm) (elt categoryForm 0) categoryForm))
  (|compilerMessage| (list '|Adding | domainName '| modemaps|))
  (setq env (|putDomainsInScope| domainName env))
  (setq |$base| 4)
  (dolist (term fnAlist)
    (setq lhs (car term))
    (setq op (caar term))
    (setq sig (cadar term))
    (setq cond (cadr term))
    (setq fnsel (caddr term))
    (setq u (|assoc| (subst '|Rep| domainName lhs :test #'equal) repFnAlist))
    (if (and u (null (redefinedList op functorBody)))
      (setq env (|addModemap| op domainName sig cond (caddr u) env))
      (setq env (|addModemap| op domainName sig cond fnsel env))))
  env)))
```

———-

6.4 Maintaining Modemaps

6.4.1 defun addModemapKnown

[addModemap0 p252]
[$e p??]
[$insideCapsuleFunctionIfTrue p??]
[$CapsuleModemapFrame p??]

— defun addModemapKnown —

```
(defun |addModemapKnown| (op mc sig pred fn |$e|)
```

```
 (declare (special |$e| |$CapsuleModemapFrame| |$insideCapsuleFunctionIfTrue|))
  (if (eq |$insideCapsuleFunctionIfTrue| t)
   (progn
     (setq |$CapsuleModemapFrame|
      (|addModemap0| op mc sig pred fn |$CapsuleModemapFrame|))
     |$e|)
   (|addModemap0| op mc sig pred fn |$e|)))
```

———-

6.4.2 defun addModemap

[addModemap0 p252]
[knownInfo p??]
[$e p??]
[$InteractiveMode p??]
[$insideCapsuleFunctionIfTrue p??]
[$CapsuleModemapFrame p??]
[$CapsuleModemapFrame p??]

— defun addModemap —

```
(defun |addModemap| (op mc sig pred fn |$e|)
 (declare (special |$e| |$CapsuleModemapFrame| |$InteractiveMode|
                   |$insideCapsuleFunctionIfTrue|))
  (cond
    (|$InteractiveMode| |$e|)
    (t
     (when (|knownInfo| pred) (setq pred t))
     (cond
       ((eq |$insideCapsuleFunctionIfTrue| t)
        (setq |$CapsuleModemapFrame|
          (|addModemap0| op mc sig pred fn |$CapsuleModemapFrame|))
        |$e|)
       (t
        (|addModemap0| op mc sig pred fn |$e|))))))
```

———-

6.4.3 defun addModemap0

[addEltModemap p245]
[addModemap1 p253]
[$functorForm p??]

— defun addModemap0 —

```
(defun |addModemap0| (op mc sig pred fn env)
 (declare (special |$functorForm|))
 (cond
  ((and (consp |$functorForm|)
```

```
        (eq (qfirst |$functorForm|) '|CategoryDefaults|)
        (eq mc '$))
    env)
  ((or (eq op '|elt|) (eq op '|setelt|))
    (|addEltModemap| op mc sig pred fn env))
  (t (|addModemap1| op mc sig pred fn env))))
```

———-

6.4.4 defun addModemap1

[getProplist p??]
[mkNewModemapList p246]
[lassoc p??]
[augProplist p??]
[unErrorRef p??]
[addBinding p??]

— defun addModemap1 —

```
(defun |addModemap1| (op mc sig pred fn env)
 (let (currentProplist newModemapList newProplist newProplistp)
  (when (eq mc '|Rep|) (setq sig (subst '$ '|Rep| sig :test #'equal)))
  (setq currentProplist (or (|getProplist| op env) nil))
  (setq newModemapList
   (|mkNewModemapList| mc sig pred fn
     (lassoc '|modemap| currentProplist) env nil))
  (setq newProplist (|augProplist| currentProplist '|modemap| newModemapList))
  (setq newProplistp (|augProplist| newProplist 'fluid t))
  (|unErrorRef| op)
  (|addBinding| op newProplistp env)))
```

———-

6.5 Indirect called comp routines

In the **compExpression** function there is the code:

```
  (if (and (atom (car x)) (setq fn (getl (car x) 'special)))
    (funcall fn x m e)
    (|compForm| x m e))))
```

6.5.1 defplist compAdd plist

We set up the `compAdd` function to handle the `add` keyword by setting the `special` keyword on the `add` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|add| 'special) 'compAdd))
```

——–-

6.5.2 defun compAdd

The compAdd function expects three arguments:

1. the **form** which is an —add— specifying the domain to extend and a set of functions to be added
2. the **mode** a —Join—, which is a set of categories and domains
3. the **env** which is a list of functions and their modemaps

The bulk of the work is performed by a call to compOrCroak which compiles the functions in the add form capsule.

The compAdd function returns a triple, the result of a call to compCapsule.

1. the **compiled capsule** which is a progn form which returns the domain
2. the **mode** from the input argument
3. the **env** prepended with the signatures of the functions in the body of the add.

[comp p530]
[compSubDomain1 p321]
[nreverse0 p??]
[NRTgetLocalIndex p201]
[compTuple2Record p256]
[compOrCroak p528]
[compCapsule p256]
[/editfile p??]
[$addForm p??]
[$addFormLhs p??]
[$EmptyMode p166]
[$NRTaddForm p??]
[$packagesUsed p??]
[$functorForm p??]
[$bootStrapMode p??]

— defun compAdd —

```
(defun compAdd (form mode env)
 (let (|$addForm| |$addFormLhs| code domainForm predicate tmp3 tmp4)
 (declare (special |$addForm| |$addFormLhs| |$EmptyMode| |$NRTaddForm|
                   |$packagesUsed| |$functorForm| |$bootStrapMode| /editfile))
  (setq |$addForm| (second form))
  (cond
   ((eq |$bootStrapMode| t)
    (cond
     ((and (consp |$addForm|) (eq (qfirst |$addForm|) '|@Tuple|))
       (setq code nil))
     (t
       (setq tmp3 (|comp| |$addForm| mode env))
       (setq code (first tmp3))
```

```
      (setq mode (second tmp3))
      (setq env (third tmp3)) tmp3))
   (list
     (list 'cond
       (list '|$bootStrapMode| code)
        (list 't
         (list '|systemError|
          (list 'list (mkq (car |$functorForm|)) "from"
                (mkq (|namestring| /editfile))
                "needs to be compiled"))))
        mode env))
  (t
   (setq |$addFormLhs| |$addForm|)
   (cond
    ((and (consp |$addForm|) (eq (qfirst |$addForm|) '|SubDomain|)
          (consp (qrest |$addForm|)) (consp (qcddr |$addForm|))
          (eq (qcdddr |$addForm|) nil))
      (setq domainForm (second |$addForm|))
      (setq predicate (third |$addForm|))
      (setq |$packagesUsed| (cons domainForm |$packagesUsed|))
      (setq |$NRTaddForm| domainForm)
      (|NRTgetLocalIndex| domainForm)
      ; need to generate slot for add form since all $ go-get
      ; slots will need to access it
      (setq tmp3 (|compSubDomain1| domainForm predicate mode env))
      (setq |$addForm| (first tmp3))
      (setq env (third tmp3)) tmp3)
    (t
     (setq |$packagesUsed|
      (if (and (consp |$addForm|) (eq (qfirst |$addForm|) '|@Tuple|))
        (append (qrest |$addForm|) |$packagesUsed|)
        (cons |$addForm| |$packagesUsed|)))
     (setq |$NRTaddForm| |$addForm|)
     (setq tmp3
      (cond
       ((and (consp |$addForm|) (eq (qfirst |$addForm|) '|@Tuple|))
        (setq |$NRTaddForm|
         (cons '|@Tuple|
          (dolist (x (cdr |$addForm|) (nreverse0 tmp4))
           (push (|NRTgetLocalIndex| x) tmp4))))
        (|compOrCroak| (|compTuple2Record| |$addForm|) |$EmptyMode| env))
       (t
       (|compOrCroak| |$addForm| |$EmptyMode| env))))
     (setq |$addForm| (first tmp3))
     (setq env (third tmp3))
     tmp3))
   (|compCapsule| (third form) mode env)))))
```

———-

6.5.3 defun compTuple2Record

— defun compTuple2Record —

```
(defun |compTuple2Record| (u)
 (let ((i 0))
  (cons '|Record|
   (loop for x in (rest u)
    collect (list '|:| (incf i) x)))))
```

———

6.5.4 defplist compCapsule plist

We set up the `compCapsule` function to handle the `capsule` keyword by setting the `special` keyword on the `capsule` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get 'capsule 'special) '|compCapsule|))
```

———

6.5.5 defun compCapsule

[bootStrapError p204]
[compCapsuleInner p257]
[addDomain p233]
[editfile p??]
[$insideExpressionIfTrue p??]
[$functorForm p??]
[$bootStrapMode p??]

— defun compCapsule —

```
(defun |compCapsule| (form mode env)
 (let (|$insideExpressionIfTrue| itemList)
 (declare (special |$insideExpressionIfTrue| |$functorForm| /editfile
                   |$bootStrapMode|))
  (setq itemList (cdr form))
  (cond
   ((eq |$bootStrapMode| t)
     (list (|bootStrapError| |$functorForm| /editfile) mode env))
   (t
    (setq |$insideExpressionIfTrue| nil)
    (|compCapsuleInner| itemList mode (|addDomain| '$ env))))))
```

———

6.5.6 defun compCapsuleInner

[addInformation p??]
[compCapsuleItems p258]
[processFunctor p257]
[mkpf p??]
[$getDomainCode p??]
[$signature p??]
[$form p??]
[$addForm p??]
[$insideCategoryPackageIfTrue p??]
[$insideCategoryIfTrue p??]
[$functorLocalParameters p??]

— defun compCapsuleInner —

```
(defun |compCapsuleInner| (form mode env)
 (let (localParList data code)
 (declare (special |$getDomainCode| |$signature| |$form| |$addForm|
                   |$insideCategoryPackageIfTrue| |$insideCategoryIfTrue|
                   |$functorLocalParameters|))
  (setq env (|addInformation| mode env))
  (setq data (cons 'progn form))
  (setq env (|compCapsuleItems| form nil env))
  (setq localParList |$functorLocalParameters|)
  (when |$addForm| (setq data (list '|add| |$addForm| data)))
  (setq code
   (if (and |$insideCategoryIfTrue| (null |$insideCategoryPackageIfTrue|))
    data
    (|processFunctor| |$form| |$signature| data localParList env)))
  (cons (mkpf (append |$getDomainCode| (list code)) 'progn) (list mode env))))
```

———-

6.5.7 defun processFunctor

[error p??]
[buildFunctor p??]

— defun processFunctor —

```
(defun |processFunctor| (form signature data localParList e)
  (cond
    ((and (consp form) (eq (qrest form) nil)
          (eq (qfirst form) '|CategoryDefaults|))
     (|error| '|CategoryDefaults is a reserved name|))
    (t (|buildFunctor| form signature data localParList e))))
```

———-

6.5.8 defun compCapsuleItems

The variable data appears to be unbound at runtime. Optimized code won't check for this but interpreted code fails. We should PROVE that data is unbound at runtime but have not done so yet. Rather than remove the code entirely (since there MIGHT be a path where it is used) we check for the runtime bound case and assign `$myFunctorBody` if data has a value.

The compCapsuleInner function in this file LOOKS like it sets data and expects code to manipulate the assigned data structure. Since we can't be sure we take the least disruptive course of action.

[compSingleCapsuleItem p258]
[$top-level p??]
[$myFunctorBody p??]
[$signatureOfForm p??]
[$suffix p??]
[$e p??]
[$pred p??]
[$e p??]

— **defun compCapsuleItems** —

```
(defun |compCapsuleItems| (itemlist |$predl| |$e|)
 (declare (special |$predl| |$e|))
 (let ($top_level |$myFunctorBody| |$signatureOfForm| |$suffix|)
 (declare (special $top_level |$myFunctorBody| |$signatureOfForm| |$suffix|))
  (setq $top_level nil)
  (setq |$myFunctorBody| nil)
  (when (boundp '|data|) (setq |$myFunctorBody| |data|))
  (setq |$signatureOfForm| nil)
  (setq |$suffix| 0)
  (loop for item in itemlist do
   (setq |$e| (|compSingleCapsuleItem| item |$predl| |$e|)))
  |$e|))
```

———

6.5.9 defun compSingleCapsuleItem

[doit p??]
[$pred p??]
[$e p??]
[macroExpandInPlace p167]

— **defun compSingleCapsuleItem** —

```
(defun |compSingleCapsuleItem| (item |$predl| |$e|)
 (declare (special |$predl| |$e|))
  (|doIt| (|macroExpandInPlace| item |$e|) |$predl|)
  |$e|)
```

———

6.5.10 defun doIt

[lastnode p**??**]
[compSingleCapsuleItem p258]
[isDomainForm p319]
[stackWarning p**??**]
[doIt p259]
[compOrCroak p528]
[stackSemanticError p**??**]
[bright p**??**]
[member p**??**]
[—isFunctor p**??**]
[insert p**??**]
[opOf p**??**]
[get p**??**]
[NRTgetLocalIndex p201]
[sublis p**??**]
[compOrCroak p528]
[sayBrightly p**??**]
[formatUnabbreviated p**??**]
[doItIf p262]
[isMacro p264]
[put p**??**]
[cannotDo p**??**]
[$predl p**??**]
[$e p**??**]
[$EmptyMode p166]
[$NonMentionableDomainNames p**??**]
[$functorLocalParameters p**??**]
[$functorsUsed p**??**]
[$packagesUsed p**??**]
[$NRTopt p**??**]
[$Representation p**??**]
[$LocalDomainAlist p**??**]
[$QuickCode p**??**]
[$signatureOfForm p**??**]
[$genno p**??**]
[$e p**??**]
[$functorLocalParameters p**??**]
[$functorsUsed p**??**]
[$packagesUsed p**??**]
[$Representation p**??**]
[$LocalDomainAlist p**??**]

— defun doIt —

```
(defun |doIt| (item |$predl|)
 (declare (special |$predl|))
 (prog ($genno x rhs lhsp lhs rhsp rhsCode z tmp1 tmp2 tmp6 op body tt
        functionPart u code)
 (declare (special $genno |$e| |$EmptyMode| |$signatureOfForm|
```

```
                 |$QuickCode| |$LocalDomainAlist| |$Representation|
                 |$NRTopt| |$packagesUsed| |$functorsUsed|
                 |$functorLocalParameters| |$NonMentionableDomainNames|))
(setq $genno 0)
(cond
 ((and (consp item) (eq (qfirst item) 'seq) (consp (qrest item))
       (progn (setq tmp6 (reverse (qrest item))) t)
       (consp tmp6) (consp (qfirst tmp6))
       (eq (qcaar tmp6) '|exit|)
       (consp (qcdar tmp6))
       (equal (qcadar tmp6) 1)
       (consp (qcddar tmp6))
       (eq (qcdddar tmp6) nil))
    (setq x (qcaddar tmp6))
    (setq z (qrest tmp6))
    (setq z (nreverse z))
    (rplaca item 'progn)
    (rplaca (lastnode item) x)
    (loop for it1 in (rest item)
     do (setq |$e| (|compSingleCapsuleItem| it1 |$predl| |$e|))))
 ((|isDomainForm| item |$e|)
  (setq u (list '|import| (cons (car item) (cdr item))))
  (|stackWarning| (list '|Use: import | (cons (car item) (cdr item))))
  (rplaca item (car u))
  (rplacd item (cdr u))
  (|doIt| item |$predl|))
 ((and (consp item) (eq (qfirst item) 'let) (consp (qrest item))
      (consp (qcddr item)))
  (setq lhs (qsecond item))
  (setq rhs (qthird item))
  (cond
   ((null (progn
          (setq tmp2 (|compOrCroak| item |$EmptyMode| |$e|))
          (and (consp tmp2)
               (progn
                (setq code (qfirst tmp2))
                (and (consp (qrest tmp2))
                     (progn
                      (and (consp (qcddr tmp2))
                           (eq (qcdddr tmp2) nil)
                           (PROGN
                            (setq |$e| (qthird tmp2))
                            t))))))))
    (|stackSemanticError|
     (cons '|cannot compile assigned value to| (|bright| lhs))
      nil))
   ((null (and (consp code) (eq (qfirst code) 'let)
              (progn
                (and (consp (qrest code))
                     (progn
                      (setq lhsp (qsecond code))
                      (and (consp (qcddr code))))))
                           (atom (qsecond code))))
    (cond
```

```
       ((and (consp code) (eq (qfirst code) 'progn))
        (|stackSemanticError|
         (list '|multiple assignment | item '| not allowed|)
         nil))
       (t
        (rplaca item (car code))
        (rplacd item (cdr code)))))
     (t
      (setq lhs lhsp)
      (cond
       ((and (null (|member| (ifcar rhs) |$NonMentionableDomainNames|))
             (null (member lhs |$functorLocalParameters|)))
        (setq |$functorLocalParameters|
         (append |$functorLocalParameters| (list lhs)))))
      (cond
       ((and (consp code) (eq (qfirst code) 'let)
             (progn
              (setq tmp2 (qrest code))
              (and (consp tmp2)
                   (progn
                    (setq tmp6 (qrest tmp2))
                    (and (consp tmp6)
                         (progn
                          (setq rhsp (qfirst tmp6))
                          t)))))
             (|isDomainForm| rhsp |$e|))
        (cond
         ((|isFunctor| rhsp)
          (setq |$functorsUsed| (|insert| (|opOf| rhsp) |$functorsUsed|))
          (setq |$packagesUsed| (|insert| (list (|opOf| rhsp))
            |$packagesUsed|))))
        (cond
         ((eq lhs '|Rep|)
          (setq |$Representation| (elt (|get| '|Rep| '|value| |$e|) 0))
          (cond
           ((eq |$NRTopt| t)
            (|NRTgetLocalIndex| |$Representation|))
           (t nil))))
        (setq |$LocalDomainAlist|
         (cons (cons lhs
          (sublis |$LocalDomainAlist| (elt (|get| lhs '|value| |$e|) 0)))
           |$LocalDomainAlist|))))
      (cond
       ((and (consp code) (eq (qfirst code) 'let))
        (rplaca item (if |$QuickCode| 'qsetrefv 'setelt))
        (setq rhsCode rhsp)
        (rplacd item (list '$ (|NRTgetLocalIndex| lhs) rhsCode)))
       (t
        (rplaca item (car code))
        (rplacd item (cdr code)))))))
   ((and (consp item) (eq (qfirst item) '|:|) (consp (qrest item))
         (consp (qcddr item)) (eq (qcdddr item) nil))
    (setq tmp1 (|compOrCroak| item |$EmptyMode| |$e|))
    (setq |$e| (caddr tmp1))
```

```
    tmp1)
   ((and (consp item) (eq (qfirst item) '|import|))
    (loop for dom in (qrest item)
     do (|sayBrightly| (cons "   importing " (|formatUnabbreviated| dom))))
    (setq tmp1 (|compOrCroak| item |$EmptyMode| |$e|))
    (setq |$e| (caddr tmp1))
    (rplaca item 'progn)
    (rplacd item nil))
   ((and (consp item) (eq (qfirst item) 'if))
    (|doItIf| item |$predl| |$e|))
   ((and (consp item) (eq (qfirst item) '|where|) (consp (qrest item)))
    (|compOrCroak| item |$EmptyMode| |$e|))
   ((and (consp item) (eq (qfirst item) 'mdef))
    (setq tmp1 (|compOrCroak| item |$EmptyMode| |$e|))
    (setq |$e| (caddr tmp1)) tmp1)
   ((and (consp item) (eq (qfirst item) 'def) (consp (qrest item))
         (consp (qsecond item)))
    (setq op (qcaadr item))
    (cond
     ((setq body (|isMacro| item |$e|))
      (setq |$e| (|put| op '|macro| body |$e|)))
     (t
      (setq tt (|compOrCroak| item |$EmptyMode| |$e|))
      (setq |$e| (caddr tt))
      (rplaca item '|CodeDefine|)
      (rplacd (cadr item) (list |$signatureOfForm|))
      (setq functionPart (list '|dispatchFunction| (car tt)))
      (rplaca (cddr item) functionPart)
      (rplacd (cddr item) nil))))
   ((setq u (|compOrCroak| item |$EmptyMode| |$e|))
     (setq code (car u))
     (setq |$e| (caddr u))
     (rplaca item (car code))
     (rplacd item (cdr code)))
   (t (|cannotDo|)))))
```

———

6.5.11 defun doItIf

[comp p530]
[userError p??]
[compSingleCapsuleItem p258]
[getSuccessEnvironment p293]
[localExtras p??]
[rplaca p??]
[rplacd p??]
[$e p??]
[$functorLocalParameters p??]
[$predl p??]
[$e p??]
[$functorLocalParameters p??]

[$getDomainCode p??]
[$Boolean p??]

— defun doItIf —

```
(defun |doItIf| (item |$predl| |$e|)
 (declare (special |$predl| |$e|))
 (labels (
  (localExtras (oldFLP)
   (let (oldFLPp flp1 gv ans nils n)
   (declare (special |$functorLocalParameters| |$getDomainCode|))
    (unless (eq oldFLP |$functorLocalParameters|)
     (setq flp1 |$functorLocalParameters|)
     (setq oldFLPp oldFLP)
     (setq n 0)
     (loop while oldFLPp
      do
       (setq oldFLPp (cdr oldFLPp))
       (setq n (1+ n)))
     (setq nils (setq ans nil))
     (loop for u in flp1
      do
       (if (or (atom u)
               (let (result)
                (loop for v in |$getDomainCode|
                 do
                 (setq result (or result
                  (and (consp v) (consp (qrest v))
                       (equal (qsecond v) u)))))
                result))
  ; Now we have to add code to compile all the elements of
  ; functorLocalParameters that were added during the conditional compilation
        (setq nils (cons u nils))
        (progn
         (setq gv (gensym))
         (setq ans (cons (list 'let gv u) ans))
         (setq nils (CONS gv nils))))
       (setq n (1+ n)))
     (setq |$functorLocalParameters| (append oldFLP (nreverse nils)))
     (nreverse ans)))))
 (let (p x y olde tmp1 pp xp oldFLP yp)
 (declare (special |$functorLocalParameters| |$Boolean|))
   (setq p (second item))
   (setq x (third item))
   (setq y (fourth item))
   (setq olde |$e|)
   (setq tmp1
    (or (|comp| p |$Boolean| |$e|)
        (|userError| (list "not a Boolean:" p))))
   (setq pp (first tmp1))
   (setq |$e| (third tmp1))
   (setq oldFLP |$functorLocalParameters|)
   (unless (eq x '|noBranch|)
     (|compSingleCapsuleItem| x |$predl| (|getSuccessEnvironment| p |$e|))
```

```
     (setq xp (localExtras oldFLP)))
   (setq oldFLP |$functorLocalParameters|)
   (unless (eq y '|noBranch|)
     (|compSingleCapsuleItem| y |$predl| (|getInverseEnvironment| p olde))
     (setq yp (localExtras oldFLP)))
   (rplaca item 'cond)
   (rplacd item (list (cons pp (cons x xp)) (cons ''t (cons y yp)))))))
```

———-

6.5.12 defun isMacro

[get p??]

— defun isMacro —

```
(defun |isMacro| (x env)
 (let (op args signature body)
  (when
   (and (consp x) (eq (qfirst x) 'def) (consp (qrest x))
        (consp (qsecond x)) (consp (qcddr x))
        (consp (qcdddr x))
        (consp (qcddddr x))
        (eq (qrest (qcddddr x)) nil))
     (setq op (qcaadr x))
     (setq args (qcdadr x))
     (setq signature (qthird x))
     (setq body (qfirst (qcddddr x)))
     (when
      (and (null (|get| op '|modemap| env))
           (null args)
           (null (|get| op '|mode| env))
           (consp signature)
           (eq (qrest signature) nil)
           (null (qfirst signature)))
       body))))
```

———-

6.5.13 defplist compCase plist

We set up the `compCase` function to handle the `case` keyword by setting the `special` keyword on the `case` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|case| 'special) '|compCase|))
```

———-

6.5.14 defun compCase

Will the jerk who commented out these two functions please NOT do so again. These functions ARE needed, and case can NOT be done by modemap alone. The reason is that A case B requires to take A evaluated, but B unevaluated. Therefore a special function is required. You may have thought that you had tested this on "failed" etc., but "failed" evaluates to it's own mode. Try it on x case $ next time.

An angry JHD - August 15th., 1984 [addDomain p233]
[compCase1 p265]
[coerce p325]

— defun compCase —

```
(defun |compCase| (form mode env)
 (let (mp td)
  (setq mp (third form))
  (setq env (|addDomain| mp env))
  (when (setq td (|compCase1| (second form) mp env)) (|coerce| td mode))))
```

———-

6.5.15 defun compCase1

[comp p530]
[getModemapList p243]
[nreverse0 p??]
[modeEqual p335]
[$Boolean p??]
[$EmptyMode p166]

— defun compCase1 —

```
(defun |compCase1| (form mode env)
 (let (xp mp ep map tmp3 tmp5 tmp6 u fn)
 (declare (special |$Boolean| |$EmptyMode|))
  (when (setq tmp3 (|comp| form |$EmptyMode| env))
   (setq xp (first tmp3))
   (setq mp (second tmp3))
   (setq ep (third tmp3))
   (when
    (setq u
     (dolist (modemap (|getModemapList| '|case| 2 ep) (nreverse0 tmp5))
        (setq map (first modemap))
        (when
          (and (consp map) (consp (qrest map)) (consp (qcddr map))
                (consp (qcdddr map))
                (eq (qcddddr map) nil)
                (|modeEqual| (fourth map) mode)
                (|modeEqual| (third map) mp))
            (push (second modemap) tmp5))))
    (when
     (setq fn
```

```
      (dolist (onepair u tmp6)
        (when (first onepair) (setq tmp6 (or tmp6 (second onepair))))))
      (list (list '|call| fn xp) |$Boolean| ep))))))
```

———-

6.5.16 defplist compCat plist

We set up the `compCat` function to handle the `Record` keyword by setting the `special` keyword on the `Record` symbol property list.

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|Record| 'special) '|compCat|))
```

———-

6.5.17 defplist compCat plist

We set up the `compCat` function to handle the `Mapping` keyword by setting the `special` keyword on the `Mapping` symbol property list.

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|Mapping| 'special) '|compCat|))
```

———-

6.5.18 defplist compCat plist

We set up the `compCat` function to handle the `Union` keyword by setting the `special` keyword on the `Union` symbol property list.

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|Union| 'special) '|compCat|))
```

———-

6.5.19 defun compCat

[getl p??]

— **defun compCat** —

```
(defun |compCat| (form mode env)
 (declare (ignore mode))
 (let (functorName fn tmp1 tmp2 funList op sig catForm)
  (setq functorName (first form))
  (when (setq fn (getl functorName '|makeFunctionList|))
```

```
   (setq tmp1 (funcall fn form form env))
   (setq funList (first tmp1))
   (setq env (second tmp1))
   (setq catForm
    (list '|Join| '(|SetCategory|)
     (cons 'category
      (cons '|domain|
       (dolist (item funList (nreverse0 tmp2))
        (setq op (first item))
        (setq sig (second item))
        (unless (eq op '=) (push (list 'signature op sig) tmp2)))))))
   (list form catForm env))))
```

———-

6.5.20 defplist compCategory plist

We set up the `compCategory` function to handle the `category` keyword by setting the `special` keyword on the `category` symbol property list.

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'category 'special) '|compCategory|))
```

———-

6.5.21 defun compCategory

[resolve p334]
[compCategoryItem p268]
[mkExplicitCategoryFunction p269]
[systemErrorHere p??]
[$sigList p??]
[$atList p??]
[$top-level p??]
[$sigList p??]
[$atList p??]

— **defun compCategory** —

```
(defun |compCategory| (form mode env)
 (let ($top_level |$sigList| |$atList| domainOrPackage z rep)
 (declare (special $top_level |$sigList| |$atList|))
  (setq $top_level t)
  (cond
   ((and
      (equal (setq mode (|resolve| mode (list '|Category|)))
             (list '|Category|))
      (consp form)
      (eq (qfirst form) 'category)
      (consp (qrest form)))
```

```
    (setq domainOrPackage (second form))
    (setq z (qcddr form))
    (setq |$sigList| nil)
    (setq |$atList| nil)
    (dolist (x z) (|compCategoryItem| x nil))
    (setq rep
      (|mkExplicitCategoryFunction| domainOrPackage |$sigList| |$atList|))
    (list rep mode env))
   (t
    (|systemErrorHere| "compCategory")))))
```

———-

6.5.22 defun compCategoryItem

[compCategoryItem p268]
[mkpf p??]
[$sigList p??]
[$atList p??]

— defun compCategoryItem —

```
(defun |compCategoryItem| (x predl)
 (let (p e a b c predlp pred y z op sig)
 (declare (special |$sigList| |$atList|))
 (cond
  ((null x) nil)
; 1. if x is a conditional expression, recurse; otherwise, form the predicate
  ((and (consp x) (eq (qfirst x) 'cond)
        (consp (qrest x)) (eq (qcddr x) nil)
        (consp (qsecond x))
        (consp (qcdadr x))
        (eq (qcddadr x) nil))
     (setq p (qcaadr x))
     (setq e (qcadadr x))
     (setq predlp (cons p predl))
     (cond
      ((and (consp e) (eq (qfirst e) 'progn))
        (setq z (qrest e))
        (dolist (y z) (|compCategoryItem| y predlp)))
      (t (|compCategoryItem| e predlp))))
  ((and (consp x) (eq (qfirst x) 'if) (consp (qrest x))
        (consp (qcddr x)) (consp (qcdddr x))
        (eq (qcddddr x) nil))
     (setq a (qsecond x))
     (setq b (qthird x))
     (setq c (qfourth x))
     (setq predlp (cons a predl))
     (unless (eq b '|noBranch|)
      (cond
       ((and (consp b) (eq (qfirst b) 'progn))
        (setq z (qrest b))
        (dolist (y z) (|compCategoryItem| y predlp)))
```

```
       (t (|compCategoryItem| b predlp))))
     (cond
      ((eq c '|noBranch|) nil)
      (t
       (setq predlp (cons (list '|not| a) predl))
       (cond
        ((and (consp c) (eq (qfirst c) 'progn))
         (setq z (qrest c))
         (dolist (y z) (|compCategoryItem| y predlp)))
        (t (|compCategoryItem| c predlp))))))
  (t
   (setq pred (if predl (mkpf predl 'and) t))
   (cond
; 2. if attribute, push it and return
     ((and (consp x) (eq (qfirst x) 'attribute)
           (consp (qrest x)) (eq (qcddr x) nil))
       (setq y (qsecond x))
       (push (mkq (list y pred)) |$atList|))
; 3. it may be a list, with PROGN as the CAR, and some information as the CDR
     ((and (consp x) (eq (qfirst x) 'progn))
       (setq z (qrest x))
       (dolist (u z) (|compCategoryItem| u predl)))
     (t
; 4. otherwise, x gives a signature for a single operator name or a list of
; names; if a list of names, recurse
      (cond ((eq (car x) 'signature) (car x)))
      (setq op (cadr x))
      (setq sig (cddr x))
      (cond
       ((null (atom op))
        (dolist (y op)
          (|compCategoryItem| (cons 'signature (cons y sig)) predl)))
       (t
; 5. branch on a single type or a signature %with source and target
        (push (mkq (list (cdr x) pred)) |$sigList|)))))))))
```

———-

6.5.23 defun mkExplicitCategoryFunction

[mkq p??]
[union p??]
[mustInstantiate p270]
[remdup p??]
[identp p??]
[wrapDomainSub p270]

— defun mkExplicitCategoryFunction —

```
(defun |mkExplicitCategoryFunction| (domainOrPackage sigList atList)
 (let (body sig parameters)
  (setq body
```

```
   (list '|mkCategory| (mkq domainOrPackage)
      (cons 'list (reverse sigList))
      (cons 'list (reverse atList))
      (mkq
        (let (result)
         (loop for item in sigList
          do
           (setq sig (car (cdaadr item)))
           (setq result
             (|union| result
               (loop for d in sig
                when (|mustInstantiate| d)
                collect d))))
        result))
      nil))
   (setq parameters
    (remdup
     (let (result)
      (loop for item in sigList
       do
        (setq sig (car (cdaadr item)))
        (setq result
         (append result
          (loop for x in sig
           when (and (identp x) (not (eq x '$)))
           collect x))))
      result)))
   (|wrapDomainSub| parameters body)))
```

———-

6.5.24 defun mustInstantiate

[getl p??]
[$DummyFunctorNames p??]

— defun mustInstantiate —

```
(defun |mustInstantiate| (d)
 (declare (special |$DummyFunctorNames|))
  (and (consp d)
       (null (or (member (qfirst d) |$DummyFunctorNames|)
                 (getl (qfirst d) '|makeFunctionList|)))))
```

———-

6.5.25 defun wrapDomainSub

— defun wrapDomainSub —

```
(defun |wrapDomainSub| (parameters x)
```

```
 (list '|DomainSubstitutionMacro| parameters x))
```

———-

6.5.26 defplist compColon plist

We set up the `compColon` function to handle the : keyword by setting the `special` keyword on the : symbol property list.

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|:| 'special) '|compColon|))
```

———-

6.5.27 defun compColon

[compColonInside p536]
[assoc p??]
[getDomainsInScope p235]
[isDomainForm p319]
[compColon member (vol5)]
[addDomain p233]
[isCategoryForm p??]
[unknownTypeError p234]
[compColon p271]
[eqsubstlist p??]
[take p??]
[length p??]
[nreverse0 p??]
[getmode p??]
[systemErrorHere p??]
[put p??]
[makeCategoryForm p274]
[genSomeVariable p??]
[$lhsOfColon p??]
[$noEnv p??]
[$insideFunctorIfTrue p??]
[$bootStrapMode p??]
[$FormalMapVariableList p249]
[$insideCategoryIfTrue p??]
[$insideExpressionIfTrue p??]

— **defun compColon** —

```
(defun |compColon| (form mode env)
  (let (|$lhsOfColon| argf argt tprime mprime r td op argl newTarget a
        signature tmp2 catform tmp3 g2 g5)
  (declare (special |$lhsOfColon| |$noEnv| |$insideFunctorIfTrue|
                    |$bootStrapMode| |$FormalMapVariableList|
```

```
                 |$insideCategoryIfTrue| |$insideExpressionIfTrue|))
    (setq argf (second form))
    (setq argt (third form))
    (if |$insideExpressionIfTrue|
      (|compColonInside| argf mode env argt)
      (progn
        (setq |$lhsOfColon| argf)
        (setq argt
         (cond
          ((and (atom argt)
                (setq tprime (|assoc| argt (|getDomainsInScope| env))))
            tprime)
          ((and (|isDomainForm| argt env) (null |$insideCategoryIfTrue|))
            (unless (|member| argt (|getDomainsInScope| env))
               (setq env (|addDomain| argt env)))
            argt)
          ((or (|isDomainForm| argt env) (|isCategoryForm| argt env))
            argt)
          ((and (consp argt) (eq (qfirst argt) '|Mapping|)
                (progn
                  (setq tmp2 (qrest argt))
                  (and (consp tmp2)
                       (progn
                        (setq mprime (qfirst tmp2))
                        (setq r (qrest tmp2))
                        t))))
            argt)
          (t
            (|unknownTypeError| argt)
            argt)))
        (cond
         ((eq (car argf) 'listof)
           (dolist (x (cdr argf) td)
             (setq td (|compColon| (list '|:| x argt) mode env))
             (setq env (third td))))
         (t
          (setq env
           (cond
            ((and (consp argf)
                  (progn
                   (setq op (qfirst argf))
                   (setq argl (qrest argf))
                   t)
                  (null (and (consp argt) (eq (qfirst argt) '|Mapping|))))
             (setq newTarget
              (eqsubstlist (take (|#| argl) |$FormalMapVariableList|)
              (dolist (x argl (nreverse0 g2))
                (setq g2
                 (cons
                  (cond
                   ((and (consp x) (eq (qfirst x) '|:|)
                         (progn
                          (setq tmp2 (qrest x))
                          (and (consp tmp2)
```

```
                        (progn
                         (setq a (qfirst tmp2))
                         (setq tmp3 (qrest tmp2))
                         (and (consp tmp3)
                              (eq (qrest tmp3) nil)
                              (progn
                               (setq mode (qfirst tmp3))
                               t))))))
                a)
               (t x))
        g2)))
       argt))
      (setq signature
       (cons '|Mapping|
        (cons newTarget
         (dolist (x argl (nreverse0 g5))
           (setq g5
            (cons
             (cond
              ((and (consp x) (eq (qfirst x) '|:|))
               (progn
                (setq tmp2 (qrest x))
                (and (consp tmp2)
                     (progn
                      (setq a (qfirst tmp2))
                      (setq tmp3 (qrest tmp2))
                      (and (consp tmp3)
                           (eq (qrest tmp3) nil)
                           (progn
                            (setq mode (qfirst tmp3))
                            t))))))
                mode)
              (t
               (or (|getmode| x env)
                   (|systemErrorHere| "compColonOld"))))
              g5))))))
      (|put| op '|mode| signature env))
     (t (|put| argf '|mode| argt env))))
 (cond
   ((and (null |$bootStrapMode|) |$insideFunctorIfTrue|
         (progn
          (setq tmp2 (|makeCategoryForm| argt env))
          (and (consp tmp2)
               (progn
                (setq catform (qfirst tmp2))
                (setq tmp3 (qrest tmp2))
                (and (consp tmp3)
                     (eq (qrest tmp3) nil)
                     (progn
                       (setq env (qfirst tmp3))
                              t))))))
     (setq env
      (|put| argf '|value| (list (|genSomeVariable|) argt |$noEnv|)
       env))))
```

```
        (list '|/throwAway| (|getmode| argf env) env )))))))
```

———-

6.5.28 defun makeCategoryForm

[isCategoryForm p??]
[compOrCroak p528]
[$EmptyMode p166]

— defun makeCategoryForm —

```
(defun |makeCategoryForm| (c env)
 (let (tmp1)
 (declare (special |$EmptyMode|))
  (when (|isCategoryForm| c env)
    (setq tmp1 (|compOrCroak| c |$EmptyMode| env))
    (list (first tmp1) (third tmp1)))))
```

———-

6.5.29 defplist compCons plist

We set up the `compCons` function to handle the `cons` keyword by setting the `special` keyword on the `cons` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get 'cons 'special) '|compCons|))
```

———-

6.5.30 defun compCons

[compCons1 p274]
[compForm p541]

— defun compCons —

```
(defun |compCons| (form mode env)
  (or (|compCons1| form mode env) (|compForm| form mode env)))
```

———-

6.5.31 defun compCons1

[comp p530]
[convert p538]
[$EmptyMode p166]

— defun compCons1 —

```
(defun |compCons1| (arg mode env)
 (let (mx y my yt mp mr ytp tmp1 x td)
 (declare (special |$EmptyMode|))
  (setq x (second arg))
  (setq y (third arg))
  (when (setq tmp1 (|comp| x |$EmptyMode| env))
   (setq x (first tmp1))
   (setq mx (second tmp1))
   (setq env (third tmp1))
   (cond
    ((null y)
     (|convert| (list (list 'list x) (list '|List| mx) env ) mode))
    (t
     (when (setq yt (|comp| y |$EmptyMode| env))
      (setq y (first yt))
      (setq my (second yt))
      (setq env (third yt))
      (setq td
       (cond
        ((and (consp my) (eq (qfirst my) '|List|) (consp (qrest my)))
          (setq mp (second my))
          (when (setq mr (list '|List| (|resolve| mp mx)))
           (when (setq ytp (|convert| yt mr))
            (when (setq tmp1 (|convert| (list x mx (third ytp)) (second mr)))
             (setq x (first tmp1))
             (setq env (third tmp1))
             (cond
              ((and (consp (car ytp)) (eq (qfirst (car ytp)) 'list))
               (list (cons 'list (cons x (cdr (car ytp)))) mr env))
              (t
               (list (list 'cons x (car ytp)) mr env)))))))
        (t
         (list (list 'cons x y) (list '|Pair| mx my) env ))))
      (|convert| td mode)))))))
```

———

6.5.32 defplist compConstruct plist

We set up the `compConstruct` function to handle the `construct` keyword by setting the `special` keyword on the `construct` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|construct| 'special) '|compConstruct|))
```

———

6.5.33 defun compConstruct

[modeIsAggregateOf p??]
[compList p540]
[convert p538]
[compForm p541]
[compVector p324]
[getDomainsInScope p235]

— defun compConstruct —

```
(defun |compConstruct| (form mode env)
 (let (z y td tp)
  (setq z (cdr form))
  (cond
   ((setq y (|modeIsAggregateOf| '|List| mode env))
    (if (setq td (|compList| z (list '|List| (cadr y)) env))
      (|convert| td mode)
      (|compForm| form mode env)))
   ((setq y (|modeIsAggregateOf| '|Vector| mode env))
    (if (setq td (|compVector| z (list '|Vector| (cadr y)) env))
      (|convert| td mode)
      (|compForm| form mode env)))
   ((setq td (|compForm| form mode env)) td)
   (t
    (dolist (d (|getDomainsInScope| env))
     (cond
      ((and (setq y (|modeIsAggregateOf| '|List| d env))
            (setq td (|compList| z (list '|List| (cadr y)) env))
            (setq tp (|convert| td mode)))
       (return tp))
      ((and (setq y (|modeIsAggregateOf| '|Vector| d env))
            (setq td (|compVector| z (list '|Vector| (cadr y)) env))
            (setq tp (|convert| td mode)))
        (return tp))))))))
```

———-

6.5.34 defplist compConstructorCategory plist

We set up the `compConstructorCategory` function to handle the `ListCategory` keyword by setting the `special` keyword on the `ListCategory` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|ListCategory| 'special) '|compConstructorCategory|))
```

———-

6.5.35 defplist compConstructorCategory plist

We set up the `compConstructorCategory` function to handle the `RecordCategory` keyword by setting the `special` keyword on the `RecordCategory` symbol property list.

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|RecordCategory| 'special) '|compConstructorCategory|))
```

———

6.5.36 defplist compConstructorCategory plist

We set up the `compConstructorCategory` function to handle the `UnionCategory` keyword by setting the `special` keyword on the `UnionCategory` symbol property list.

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|UnionCategory| 'special) '|compConstructorCategory|))
```

———

6.5.37 defplist compConstructorCategory plist

We set up the `compConstructorCategory` function to handle the `VectorCategory` keyword by setting the `special` keyword on the `VectorCategory` symbol property list.

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|VectorCategory| 'special) '|compConstructorCategory|))
```

———

6.5.38 defun compConstructorCategory

[resolve p334]
[$Category p??]

— **defun compConstructorCategory** —

```
(defun |compConstructorCategory| (form mode env)
 (declare (special |$Category|))
 (list form (|resolve| |$Category| mode) env))
```

———

6.5.39 defun getAbbreviation

[constructor? p??]
[assq p??]

[mkAbbrev p278]
[rplac p??]
[$abbreviationTable p??]
[$abbreviationTable p??]

— defun getAbbreviation —

```
(defun |getAbbreviation| (name c)
 (let (cname x n upc newAbbreviation)
 (declare (special |$abbreviationTable|))
  (setq cname (|constructor?| name))
  (cond
   ((setq x (assq cname |$abbreviationTable|))
    (cond
     ((setq n (assq name (cdr x)))
      (cond
       ((setq upc (assq c (cdr n)))
        (cdr upc))
       (t
        (setq newAbbreviation (|mkAbbrev| x cname))
        (rplac (cdr n) (cons (cons c newAbbreviation) (cdr n)))
        newAbbreviation)))
     (t
      (setq newAbbreviation (|mkAbbrev| x x))
      (rplac (cdr x)
             (cons (cons name (list (cons c newAbbreviation))) (cdr x)))
      newAbbreviation)))
   (t
    (setq |$abbreviationTable|
     (cons (list cname (list name (cons c cname))) |$abbreviationTable|))
    cname))))
```

———-

6.5.40 defun mkAbbrev

[addSuffix p278]
[alistSize p279]

— defun mkAbbrev —

```
(defun |mkAbbrev| (x z)
 (|addSuffix| (|alistSize| (cdr x)) z))
```

———-

6.5.41 defun addSuffix

— defun addSuffix —

```
(defun |addSuffix| (n u)
```

```
 (let (s)
  (if (alpha-char-p (elt (setq s (princ-to-string u)) (maxindex s)))
    (intern (strconc s (princ-to-string n)))
    (internl (strconc s (princ-to-string '|;|) (princ-to-string n))))))
```

———-

6.5.42 defun alistSize

— defun alistSize —

```
(defun |alistSize| (c)
 (labels (
  (count (x level)
   (cond
    ((eql level 2)  (|#| x))
    ((null x) 0)
    (+ (count (cdar x) (1+ level))
       (count (cdr x) level)))))
 (count c 1)))
```

———-

6.5.43 defun getSignatureFromMode

[getmode p??]
[opOf p??]
[length p??]
[stackAndThrow p??]
[eqsubstlist p??]
[take p??]
[$FormalMapVariableList p249]

— defun getSignatureFromMode —

```
(defun |getSignatureFromMode| (form env)
 (let (tmp1 signature)
 (declare (special |$FormalMapVariableList|))
  (setq tmp1 (|getmode| (|opOf| form) env))
  (when (and (consp tmp1) (eq (qfirst tmp1) '|Mapping|))
   (setq signature (qrest tmp1))
   (if (not (eql (|#| form) (|#| signature)))
     (|stackAndThrow| (list '|Wrong number of arguments: | form))
     (eqsubstlist (cdr form)
       (take (|#| (cdr form)) |$FormalMapVariableList|)
       signature)))))
```

———-

6.5.44 defun getSpecialCaseAssoc

[$functorForm p??]
[$functorSpecialCases p??]

— defun getSpecialCaseAssoc —

```
(defun |getSpecialCaseAssoc| ()
 (declare (special |$functorSpecialCases| |$functorForm|))
  (loop for r in (rest |$functorForm|)
        for z in (rest |$functorSpecialCases|)
   when z
   collect (cons r z)))
```

———-

6.5.45 defun addArgumentConditions

[mkq p??]
[systemErrorHere p??]
[$true p??]
[$functionName p??]
[$body p??]
[$argumentConditionList p??]
[$argumentConditionList p??]

— defun addArgumentConditions —

```
(defun |addArgumentConditions| (|$body| |$functionName|)
 (declare (special |$body| |$functionName| |$argumentConditionList| |$true|))
 (labels (
  (fn (clist)
   (let (n untypedCondition typedCondition)
    (cond
     ((and (consp clist) (consp (qfirst clist)) (consp (qcdar clist))
           (consp (qcddar clist))
           (eq (qcdddar clist) nil))
      (setq n (qcaar clist))
      (setq untypedCondition (qcadar clist))
      (setq typedCondition (qcaddar clist))
      (list 'cond
       (list typedCondition (fn (cdr clist)))
        (list |$true|
         (list '|argumentDataError| n
          (mkq untypedCondition) (mkq |$functionName|)))))
     ((null clist) |$body|)
     (t (|systemErrorHere| "addArgumentConditions"))))))
 (if |$argumentConditionList|
   (fn |$argumentConditionList|)
   |$body|)))
```

———-

6.5.46 defun stripOffSubdomainConditions

[assoc p??]
[mkpf p??]
[$argumentConditionList p??]
[$argumentConditionList p??]

— defun stripOffSubdomainConditions —

```
(defun |stripOffSubdomainConditions| (margl argl)
 (let (pair (i 0))
 (declare (special |$argumentConditionList|))
  (loop for x in margl for arg in argl
   do (incf i)
   collect
    (cond
     ((and (consp x) (eq (qfirst x) '|SubDomain|) (consp (qrest x))
           (consp (qcddr x)) (eq (qcdddr x) nil))
       (cond
        ((setq pair (|assoc| i |$argumentConditionList|))
          (rplac (cadr pair) (mkpf (list (third x) (cadr pair)) 'and))
          (second x))
        (t
         (setq |$argumentConditionList|
          (cons (list i arg (third x)) |$argumentConditionList|))
         (second x))))
     (t x)))))
```

———-

6.5.47 defun stripOffArgumentConditions

[$argumentConditionList p??]
[$argumentConditionList p??]

— defun stripOffArgumentConditions —

```
(defun |stripOffArgumentConditions| (argl)
 (let (condition (i 0))
 (declare (special |$argumentConditionList|))
  (loop for x in argl
   do (incf i)
   collect
    (cond
     ((and (consp x) (eq (qfirst x) '|\||) (consp (qrest x))
           (consp (qcddr x)) (eq (qcdddr x) nil))
       (setq condition (subst '|#1| (second x) (third x) :test #'equal))
       (setq |$argumentConditionList|
        (cons (list i (second x) condition) |$argumentConditionList|))
       (second x))
```

```
(t x)))))
```

———-

6.5.48 defun getSignature

Try to return a signature. If there isn't one, complain and return nil. If there are more than one then remove any that are subsumed. If there is still more than one complain else return the only signature. [get p??]
[length p??]
[remdup p??]
[knownInfo p??]
[getmode p??]
[say p??]
[printSignature p??]
[SourceLevelSubsume p??]
[stackSemanticError p??]
[$e p??]

— defun getSignature —

```
(defun |getSignature| (op argModeList |$e|)
 (declare (special |$e|))
 (let (mmList pred u tmp1 dc sig sigl)
  (setq mmList (|get| op '|modemap| |$e|))
  (cond
   ((eql 1
     (|#| (setq sigl (remdup
       (loop for item in mmList
        do
         (setq dc (caar item))
         (setq sig (cdar item))
         (setq pred (caadr item))
        when (and (eq dc '$) (equal (cdr sig) argModeList) (|knownInfo| pred))
        collect sig)))))
     (car sigl))
   ((null sigl)
     (cond
      ((progn
        (setq tmp1 (setq u (|getmode| op |$e|)))
        (and (consp tmp1) (eq (qfirst tmp1) '|Mapping|)))
       (qrest tmp1))
      (t
       (say "************* USER ERROR **********")
       (say "available signatures for " op ": ")
       (cond
        ((null mmList) (say "    NONE"))
        (t
         (loop for item in mmList
          do (|printSignature| '|     | op (cdar item)))
         (|printSignature| '|NEED | op (cons '? argModeList))))
       nil)))
```

```
(t
 ; Before we complain about duplicate signatures, we should
 ;  check that we do not have for example, a partial - as
 ;  well as a total one.  SourceLevelSubsume should do this
 (loop for u in sigl do
  (loop for v in sigl
   when (null (equal u v))
   do (when (|SourceLevelSubsume| u v) (setq sigl (|delete| v sigl)))))
 (cond
  ((eql 1 (|#| sigl)) (car sigl))
  (t
   (|stackSemanticError|
    (list '|duplicate signatures for | op '|: | argModeList) nil)))))))
```

———-

6.5.49 defun checkAndDeclare

[getArgumentMode p285]
[modeEqual p335]
[put p??]
[sayBrightly p??]
[bright p??]

— **defun checkAndDeclare** —

```
(defun |checkAndDeclare| (argl form sig env)
 (let (m1 stack)
  (loop for a in argl for m in (rest sig)
   do
    (if (setq m1 (|getArgumentMode| a env))
     (if (null (|modeEqual| m1 m))
       (setq stack
        (cons '|   | (append (|bright| a)
          (cons "must have type "
           (cons m
            (cons " not "
             (cons m1
               (cons '|%l| stack)))))))))
      (setq env (|put| a '|mode| m env))))
  (when stack
   (|sayBrightly|
    (cons "   Parameters of "
     (append (|bright| (car form))
       (cons " are of wrong type:"
        (cons '|%l| stack))))))
  env))
```

———-

6.5.50 defun hasSigInTargetCategory

[getArgumentMode p285]
[remdup p??]
[length p??]
[getSignatureFromMode p279]
[stackWarning p??]
[compareMode2Arg p??]
[bright p??]
[$domainShell p??]

— defun hasSigInTargetCategory —

```
(defun |hasSigInTargetCategory| (argl form opsig env)
 (labels (
  (fn (opName sig opsig mList form)
   (declare (special |$op|))
    (and
     (and
      (and (equal opName |$op|) (equal (|#| sig) (|#| form)))
      (or (null opsig) (equal opsig (car sig))))
     (let ((result t))
      (loop for x in mList for y in (rest sig)
       do (setq result (and result (or (null x) (|modeEqual| x y)))))
      result))))
 (let (mList potentialSigList c sig)
 (declare (special |$domainShell|))
  (setq mList
   (loop for x in argl
    collect (|getArgumentMode| x env)))
  (setq potentialSigList
   (remdup
    (loop for item in (elt |$domainShell| 1)
     when (fn (caar item) (cadar item) opsig mList form)
     collect (cadar item))))
  (setq c (|#| potentialSigList))
  (cond
   ((eql 1 c) (car potentialSigList))
   ((eql 0 c)
    (when (equal (|#| (setq sig (|getSignatureFromMode| form env))) (|#| form))
      sig))
   ((> c 1)
    (setq sig (car potentialSigList))
    (|stackWarning|
     (cons '|signature of lhs not unique:|
      (append (|bright| sig) (list '|chosen|))))
    sig)
   (t nil)))))
```

———

6.5.51 defun getArgumentMode

[get p??]

— defun getArgumentMode —

```
(defun |getArgumentMode| (x e)
  (if (stringp x) x (|get| x '|mode| e)))
```

———-

6.5.52 defplist compElt plist

We set up the `compElt` function to handle the `elt` keyword by setting the `special` keyword on the `elt` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|elt| 'special) '|compElt|))
```

———-

6.5.53 defun compElt

[compForm p541]
[isDomainForm p319]
[addDomain p233]
[getModemapListFromDomain p244]
[length p??]
[stackMessage p??]
[stackWarning p??]
[convert p538]
[opOf p??]
[getDeltaEntry p??]
[$One p??]
[$Zero p??]

— defun compElt —

```
(defun |compElt| (form mode env)
 (let (aDomain anOp mmList n modemap sig pred val)
 (declare (special |$One| |$Zero|))
 (setq anOp (third form))
 (setq aDomain (second form))
 (cond
  ((null (and (consp form) (eq (qfirst form) '|elt|)
              (consp (qrest form)) (consp (qcddr form))
              (eq (qcdddr form) nil)))
   (|compForm| form mode env))
  ((eq aDomain '|Lisp|)
   (list (cond
```

```
          ((equal anOp |$Zero|) 0)
          ((equal anOp |$One|) 1)
          (t anOp))
      mode env))
  ((|isDomainForm| aDomain env)
   (setq env (|addDomain| aDomain env))
   (setq mmList (|getModemapListFromDomain| anOp 0 aDomain env))
   (setq modemap
    (progn
     (setq n (|#| mmList))
     (cond
      ((eql 1 n) (elt mmList 0))
      ((eql 0 n)
        (|stackMessage|
          (list "Operation " anOp "missing from domain: "
                aDomain nil))
        nil)
      (t
       (|stackWarning|
        (list "more than 1 modemap for: " anOp " with dc="
               aDomain " ===>" mmList ))
        (elt mmList 0)))))
   (when modemap
    (setq sig (first modemap))
    (setq pred (caadr modemap))
    (setq val (cadadr modemap))
    (unless (and (not (eql (|#| sig) 2))
                 (null (and (consp val) (eq (qfirst val) '|elt|))))
       (setq val (|genDeltaEntry| (cons (|opOf| anOp) modemap)))
       (|convert| (list (list '|call| val) (second sig) env) mode))))
  (t
    (|compForm| form mode env)))))
```

———-

6.5.54 defplist compExit plist

We set up the `compExit` function to handle the `exit` keyword by setting the `special` keyword on the `exit` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|exit| 'special) '|compExit|))
```

———-

6.5.55 defun compExit

[comp p530]
[modifyModeStack p561]
[stackMessageIfNone p??]

[$exitModeStack p??]

— defun compExit —

```
(defun |compExit| (form mode env)
 (let (exitForm index m1 u)
 (declare (special |$exitModeStack|))
  (setq index (1- (second form)))
  (setq exitForm (third form))
  (cond
   ((null |$exitModeStack|)
    (|comp| exitForm mode env))
   (t
    (setq m1 (elt |$exitModeStack| index))
    (setq u (|comp| exitForm m1 env))
    (cond
     (u
      (|modifyModeStack| (second u) index)
      (list (list '|TAGGEDexit| index u) mode env))
     (t
       (|stackMessageIfNone|
        (list '|cannot compile exit expression| exitForm '|in mode| m1))))))))
```

———-

6.5.56 defplist compHas plist

We set up the `compHas` function to handle the `has` keyword by setting the `special` keyword on the `has` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|has| 'special) '|compHas|))
```

———-

6.5.57 defun compHas

[chaseInferences p??]
[compHasFormat p288]
[coerce p325]
[$e p??]
[$e p??]
[$Boolean p??]

— defun compHas —

```
(defun |compHas| (pred mode |$e|)
 (declare (special |$e| |$Boolean|))
 (let (a b predCode)
  (setq a (second pred))
  (setq b (third pred))
```

```
  (setq |$e| (|chaseInferences| pred |$e|))
  (setq predCode (|compHasFormat| pred))
  (|coerce| (list predCode |$Boolean| |$e|) mode)))
```

———-

6.5.58 defun compHasFormat

[take p??]
[length p??]
[sublislis p??]
[comp p530]
[mkList p289]
[mkDomainConstructor p??]
[isDomainForm p319]
[$FormalMapVariableList p249]
[$EmptyMode p166]
[$e p??]
[$form p??]
[$EmptyEnvironment p??]

— defun compHasFormat —

```
(defun |compHasFormat| (pred)
 (let (olda b argl formals tmp1 a)
 (declare (special |$EmptyEnvironment| |$e| |$EmptyMode|
                   |$FormalMapVariableList| |$form|))
  (when (eq (car pred) '|has|) (car pred))
  (setq olda (second pred))
  (setq b (third pred))
  (setq argl (rest |$form|))
  (setq formals (take (|#| argl) |$FormalMapVariableList|))
  (setq a (sublislis argl formals olda))
  (setq tmp1 (|comp| a |$EmptyMode| |$e|))
  (when tmp1
   (setq a (car tmp1))
   (setq a (sublislis formals argl a))
   (cond
    ((and (consp b) (eq (qfirst b) 'attribute) (consp (qrest b))
          (eq (qcddr b) nil))
      (list '|HasAttribute| a (list 'quote (qsecond b))))
    ((and (consp b) (eq (qfirst b) 'signature) (consp (qrest b))
          (consp (qcddr b)) (eq (qcdddr b) NIL))
       (list '|HasSignature| a
         (|mkList|
          (list (MKQ (qsecond b))
           (|mkList|
            (loop for type in (qthird b)
             collect (|mkDomainConstructor| type)))))))
    ((|isDomainForm| b |$EmptyEnvironment|)
      (list 'equal a b))
    (t
```

```
    (list '|HasCategory| a (|mkDomainConstructor| b)))))))
```

———-

6.5.59 defun mkList

— defun mkList —

```
(defun |mkList| (u)
 (when u (cons 'list u)))
```

———-

6.5.60 defplist compIf plist

We set up the `compIf` function to handle the `if` keyword by setting the `special` keyword on the `if` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get 'if 'special) '|compIf|))
```

———-

6.5.61 defun compIf

[canReturn p290]
[intersectionEnvironment p??]
[compBoolean p292]
[compFromIf p290]
[resolve p334]
[coerce p325]
[quotify p??]
[$Boolean p??]

— defun compIf —

```
(defun |compIf| (form mode env)
 (labels (
  (environ (bEnv cEnv b c env)
   (cond
    ((|canReturn| b 0 0 t)
     (if (|canReturn| c 0 0 t) (|intersectionEnvironment| bEnv cEnv) bEnv))
    ((|canReturn| c 0 0 t) cEnv)
    (t env))))
  (let (a b c tmp1 xa ma Ea Einv Tb xb mb Eb Tc xc mc Ec xbp x returnEnv)
  (declare (special |$Boolean|))
   (setq a (second form))
   (setq b (third form))
```

```
   (setq c (fourth form))
   (when  (setq tmp1 (|compBoolean| a |$Boolean| env))
    (setq xa (first tmp1))
    (setq ma (second tmp1))
    (setq Ea (third tmp1))
    (setq Einv (fourth tmp1))
    (when (setq Tb (|compFromIf| b mode Ea))
     (setq xb (first Tb))
     (setq mb (second Tb))
     (setq Eb (third Tb))
     (when (setq Tc (|compFromIf| c (|resolve| mb mode) Einv))
      (setq xc (first Tc))
      (setq mc (second Tc))
      (setq Ec (third Tc))
      (when (setq xbp (|coerce| Tb mc))
       (setq x (list 'if xa (first xbp) xc))
       (setq returnEnv (environ (third xbp) Ec (first xbp) xc env))
       (list x mc returnEnv))))))))
```

———-

6.5.62 defun compFromIf

[comp p530]

— defun compFromIf —

```
(defun |compFromIf| (a m env)
  (if (eq a '|noBranch|)
    (list '|noBranch| m env)
    (|comp| a m env)))
```

———-

6.5.63 defun canReturn

[say p??]
[canReturn p290]
[systemErrorHere p??]

— defun canReturn —

```
(defun |canReturn| (expr level exitCount ValueFlag)
 (labels (
  (findThrow (gs expr level exitCount ValueFlag)
   (cond
    ((atom expr) nil)
    ((and (consp expr) (eq (qfirst expr) 'throw) (consp (qrest expr))
          (equal (qsecond expr) gs) (consp (qcddr expr))
          (eq (qcdddr expr) nil))
      t)
    ((and (consp expr) (eq (qfirst expr) 'seq))
```

```
     (let (result)
       (loop for u in (qrest expr)
        do (setq result
            (or result
             (findThrow gs u (1+ level) exitCount ValueFlag))))
        result))
    (t
     (let (result)
       (loop for u in (rest expr)
        do (setq result
            (or result
             (findThrow gs u level exitCount ValueFlag))))
        result)))))
 (let (op gs)
  (cond
   ((atom expr) (and ValueFlag (equal level exitCount)))
   ((eq (setq op (car expr)) 'quote) (and ValueFlag (equal level exitCount)))
   ((eq op '|TAGGEDexit|)
    (cond
     ((and (consp expr) (consp (qrest expr)) (consp (qcddr expr))
           (eq (qcdddr expr) nil))
       (|canReturn| (car (third expr)) level (second expr)
                    (equal (second expr) level))))
     ((and (equal level exitCount) (null ValueFlag))
       nil)
     ((eq op 'seq)
      (let (result)
       (loop for u in (rest expr)
        do (setq result (or result (|canReturn| u (1+ level) exitCount nil))))
       result))
     ((eq op '|TAGGEDreturn|) nil)
     ((eq op 'catch)
      (cond
       ((findThrow (second expr) (third expr) level
                       exitCount ValueFlag)
         t)
       (t
        (|canReturn| (third expr) level exitCount ValueFlag))))
     ((eq op 'cond)
      (cond
       ((equal level exitCount)
        (let (result)
         (loop for u in (rest expr)
          do (setq result (or result
               (|canReturn| (|last| u) level exitCount ValueFlag))))
         result))
       (t
        (let (outer)
         (loop for v in (rest expr)
          do (setq outer (or outer
               (let (inner)
                (loop for u in v
                 do (setq inner
                     (or inner
```

```
                     (findThrow gs u level exitCount ValueFlag))))
               inner))))
          outer))))
     ((eq op 'if)
       (and (consp expr) (consp (qrest expr)) (consp (qcddr expr))
            (consp (qcdddr expr))
            (eq (qcddddr expr) nil))
        (cond
         ((null (|canReturn| (second expr) 0 0 t))
           (say "IF statement can not cause consequents to be executed")
           (|pp| expr)))
          (or (|canReturn| (second expr) level exitCount nil)
              (|canReturn| (third expr) level exitCount ValueFlag)
              (|canReturn| (fourth expr) level exitCount ValueFlag)))
     ((atom op)
      (let ((result t))
       (loop for u in expr
        do (setq result
            (and result (|canReturn| u level exitCount ValueFlag))))
       result))
     ((and (consp op) (eq (qfirst op) 'xlam) (consp (qrest op))
           (consp (qcddr op)) (eq (qcdddr op) nil))
       (let ((result t))
        (loop for u in expr
         do (setq result
              (and result (|canReturn| u level exitCount ValueFlag))))
        result))
      (t (|systemErrorHere| "canReturn"))))))
```

———

6.5.64 defun compBoolean

[comp p530]
[getSuccessEnvironment p293]
[getInverseEnvironment p294]

— defun compBoolean —

```
(defun |compBoolean| (p mode env)
 (let (tmp1 pp)
  (when (setq tmp1 (OR (|comp| p mode env)))
   (setq pp (car tmp1))
   (setq mode (cadr tmp1))
   (setq env (caddr tmp1))
   (list pp mode (|getSuccessEnvironment| p env)
                 (|getInverseEnvironment| p env)))))
```

———

6.5.65 defun getSuccessEnvironment

[isDomainForm p319]
[put p??]
[identp p??]
[getProplist p??]
[comp p530]
[consProplistOf p??]
[removeEnv p??]
[addBinding p??]
[get p??]
[$EmptyEnvironment p??]
[$EmptyMode p166]

— defun getSuccessEnvironment —

```
(defun |getSuccessEnvironment| (a env)
 (let (id currentProplist tt newProplist x m)
  (declare (special |$EmptyMode| |$EmptyEnvironment|))
   (cond
    ((and (consp a) (eq (qfirst a) '|has|) (CONSP (qrest a))
          (consp (qcddr a)) (eq (qcdddr a) nil))
     (if
      (and (identp (second a)) (|isDomainForm| (third a) |$EmptyEnvironment|))
        (|put| (second a) '|specialCase| (third a) env)
        env))
    ((and (consp a) (eq (qfirst a) '|is|) (consp (qrest a))
          (consp (qcddr a)) (eq (qcdddr a) nil))
       (setq id (qsecond a))
       (setq m (qthird a))
       (cond
         ((and (identp id) (|isDomainForm| m |$EmptyEnvironment|))
          (setq env (|put| id '|specialCase| m env))
          (setq currentProplist (|getProplist| id env))
          (setq tt (|comp| m |$EmptyMode| env))
          (when tt
           (setq env (caddr tt))
           (setq newProplist
             (|consProplistOf| id currentProplist '|value|
                         (cons m (cdr (|removeEnv| tt)))))
             (|addBinding| id newProplist env)))
         (t env)))
    ((and (consp a) (eq (qfirst a) '|case|) (consp (qrest a))
          (consp (qcddr a)) (eq (qcdddr a) nil)
          (identp (qsecond a)))
      (setq x (qsecond a))
      (setq m (qthird a))
      (|put| x '|condition| (cons a (|get| x '|condition| env)) env))
    (t env))))
```

———

6.5.66 defun getInverseEnvironment

[identp p??]
[isDomainForm p319]
[put p??]
[get p??]
[member p??]
[mkpf p??]
[delete p??]
[getUnionMode p295]
[$EmptyEnvironment p??]

— **defun getInverseEnvironment** —

```
(defun |getInverseEnvironment| (a env)
 (let (op argl x m oldpred tmp1 zz newpred)
 (declare (special |$EmptyEnvironment|))
  (cond
   ((atom a) env)
   (t
    (setq op (car a))
    (setq argl (cdr a))
    (cond
     ((eq op '|has|)
       (setq x (car argl))
       (setq m (cadr argl))
       (cond
        ((and (identp x) (|isDomainForm| m |$EmptyEnvironment|))
           (|put| x '|specialCase| m env))
        (t env)))
     ((and (consp a) (eq (qfirst a) '|case|) (consp (qrest a))
           (consp (qcddr a)) (eq (qcdddr a) nil)
           (identp (qsecond a)))
       (setq x (qsecond a))
       (setq m (qthird a))
       (setq tmp1 (|get| x '|condition| env))
       (cond
        ((and tmp1 (consp tmp1) (eq (qrest tmp1) nil) (consp (qfirst tmp1))
              (eq (qcaar tmp1) 'or) (|member| a (qcdar tmp1)))
          (setq oldpred (qcdar tmp1))
          (|put| x '|condition| (list (mkpf (|delete| a oldpred) 'or)) env))
        (t
         (setq tmp1 (|getUnionMode| x env))
         (setq zz (|delete| m (qrest tmp1)))
         (loop for u in zz
          when (and (consp u) (eq (qfirst u) '|:|)
                    (consp (qrest u)) (equal (qsecond u) m))
          do (setq zz (|delete| u zz)))
         (setq newpred
          (mkpf (loop for mp in zz collect (list '|case| x mp)) 'or))
         (|put| x '|condition|
                   (cons newpred (|get| x '|condition| env)) env))))
     (t env))))))
```

———-

6.5.67 defun getUnionMode

[isUnionMode p295]
[getmode p??]

— defun getUnionMode —

```
(defun |getUnionMode| (x env)
 (let (m)
  (setq m (when (atom x) (|getmode| x env)))
  (when m (|isUnionMode| m env))))
```

———-

6.5.68 defun isUnionMode

[getmode p??]
[get p??]

— defun isUnionMode —

```
(defun |isUnionMode| (m env)
 (let (mp v tmp1)
  (cond
   ((and (consp m) (eq (qfirst m) '|Union|)) m)
   ((progn
     (setq tmp1 (setq mp (|getmode| m env)))
     (and (consp tmp1) (eq (qfirst tmp1) '|Mapping|)
          (consp (qrest tmp1)) (eq (qcddr tmp1) nil)
          (consp (qsecond tmp1))
          (eq (qcaadr tmp1) '|UnionCategory|)))
      (second mp))
   ((setq v (|get| (if (eq m '$) '|Rep| m) '|value| env))
     (when  (and (consp (car v)) (eq (qfirst (car v)) '|Union|)) (car v))))))
```

———-

6.5.69 defplist compImport plist

We set up the `compImport` function to handle the `import` keyword by setting the `special` keyword on the `import` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|import| 'special) '|compImport|))
```

———-

6.5.70 defun compImport

[addDomain p233]
[$NoValueMode p165]

— defun compImport —

```
(defun |compImport| (form mode env)
 (declare (ignore mode))
 (declare (special |$NoValueMode|))
  (dolist (dom (cdr form)) (setq env (|addDomain| dom env)))
  (list '|/throwAway| |$NoValueMode| env))
```

———-

6.5.71 defplist compIs plist

We set up the `compIs` function to handle the `is` keyword by setting the `special` keyword on the `is` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|is| 'special) '|compIs|))
```

———-

6.5.72 defun compIs

[comp p530]
[coerce p325]
[$Boolean p??]
[$EmptyMode p166]

— defun compIs —

```
(defun |compIs| (form mode env)
 (let (a b aval am tmp1 bval bm td)
 (declare (special |$Boolean| |$EmptyMode|))
  (setq a (second form))
  (setq b (third form))
  (when (setq tmp1 (|comp| a |$EmptyMode| env))
   (setq aval (first tmp1))
   (setq am (second tmp1))
   (setq env (third tmp1))
   (when (setq tmp1 (|comp| b |$EmptyMode| env))
    (setq bval (first tmp1))
    (setq bm (second tmp1))
    (setq env (third tmp1))
    (setq td (list (list '|domainEqual| aval bval) |$Boolean| env ))
    (|coerce| td mode)))))
```

———-

6.5.73 defplist compJoin plist

We set up the compJoin function to handle the Join keyword by setting the special keyword on the Join symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|Join| 'special) '|compJoin|))
```

———-

6.5.74 defun compJoin

[nreverse0 p??]
[compForMode p298]
[stackSemanticError p??]
[nreverse0 p??]
[isCategoryForm p??]
[union p??]
[compJoin,getParms p??]
[wrapDomainSub p270]
[convert p538]
[$Category p??]

— defun compJoin —

```
(defun |compJoin| (form mode env)
 (labels (
  (getParms (y env)
   (cond
    ((atom y)
     (when (|isDomainForm| y env) (list y)))
    ((and (consp y) (eq (qfirst y) 'length)
          (consp (qrest y)) (eq (qcddr y) nil))
     (list y (second y)))
    (t (list y)))) )
 (let (argl catList pl tmp3 tmp4 tmp5 body parameters catListp td)
 (declare (special |$Category|))
  (setq argl (cdr form))
  (setq catList
   (dolist (x argl (nreverse0 tmp3))
    (push (car (or (|compForMode| x |$Category| env) (return '|failed|)))
       tmp3)))
  (cond
   ((eq catList '|failed|)
    (|stackSemanticError| (list '|cannot form Join of: | argl) nil))
   (t
    (setq catListp
     (dolist (x catList (nreverse0 tmp4))
       (setq tmp4
```

```
        (cons
         (cond
          ((|isCategoryForm| x env)
           (setq parameters
            (|union|
              (dolist (y (cdr x) tmp5)
               (setq tmp5 (append tmp5 (getParms y env))))
              parameters))
           x)
          ((and (consp x) (eq (qfirst x) '|DomainSubstitutionMacro|)
                (consp (qrest x)) (consp (qcddr x))
                (eq (qcdddr x) nil))
           (setq pl (second x))
           (setq body (third x))
           (setq parameters (|union| pl parameters)) body)
          ((and (consp x) (eq (qfirst x) '|mkCategory|))
            x)
          ((and (atom x) (equal (|getmode| x env) |$Category|))
            x)
          (t
           (|stackSemanticError| (list '|invalid argument to Join: | x) nil)
           x))
         tmp4))))
    (setq td (list (|wrapDomainSub| parameters (cons '|Join| catListp))
           |$Category| env))
    (|convert| td mode)))))
```

———-

6.5.75 defun compForMode

[comp p530]
[$compForModeIfTrue p??]

— defun compForMode —

```
(defun |compForMode| (x m e)
 (let (|$compForModeIfTrue|)
 (declare (special |$compForModeIfTrue|))
  (setq |$compForModeIfTrue| t)
  (|comp| x m e)))
```

———-

6.5.76 defplist compLambda plist

We set up the `compLambda` function to handle the `+->` keyword by setting the `special` keyword on the `+->` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|+->| 'special) '|compLambda|))
```

———-

6.5.77 defun compLambda

[argsToSig p560]
[compAtSign p331]
[stackAndThrow p??]

— defun compLambda —

```
(defun |compLambda| (form mode env)
 (let (vl body tmp1 tmp2 tmp3 target args arg1 sig1 ress)
  (setq vl (second form))
  (setq body (third form))
  (cond
   ((and (consp vl) (eq (qfirst vl) '|:|)
         (progn
          (setq tmp1 (qrest vl))
          (and (consp tmp1)
               (progn
                (setq args (qfirst tmp1))
                (setq tmp2 (qrest tmp1))
                (and (consp tmp2)
                     (eq (qrest tmp2) nil)
                     (progn
                      (setq target (qfirst tmp2))
                      t))))))
     (when (and (consp args) (eq (qfirst args) '|@Tuple|))
      (setq args (qrest args)))
     (cond
      ((listp args)
       (setq tmp3 (|argsToSig| args))
       (setq arg1 (first tmp3))
       (setq sig1 (second tmp3))
       (cond
        (sig1
         (setq ress
          (compAtSign
           (list '@
            (list '+-> arg1 body)
            (cons '|Mapping| (cons target sig1))) mode env))
          ress)
        (t (|stackAndThrow| (list '|compLambda| form )))))
      (t (|stackAndThrow| (list '|compLambda| form )))))
    (t (|stackAndThrow| (list '|compLambda| form ))))))
```

———-

6.5.78 defplist compLeave plist

We set up the compLeave function to handle the leave keyword by setting the special keyword on the leave symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|leave| 'special) '|compLeave|))
```

———

6.5.79 defun compLeave

[comp p530]
[modifyModeStack p561]
[$exitModeStack p??]
[$leaveLevelStack p??]

— defun compLeave —

```
(defun |compLeave| (form mode env)
 (let (level x index u)
  (declare (special |$exitModeStack| |$leaveLevelStack|))
   (setq level (second form))
   (setq x (third form))
   (setq index
     (- (1- (|#| |$exitModeStack|)) (elt |$leaveLevelStack| (1- level))))
   (when (setq u (|comp| x (elt |$exitModeStack| index) env))
    (|modifyModeStack| (second u) index)
    (list (list '|TAGGEDexit| index u) mode env ))))
```

———

6.5.80 defplist compMacro plist

We set up the compMacro function to handle the MDEF keyword by setting the special keyword on the MDEF symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get 'mdef 'special) '|compMacro|))
```

———

6.5.81 defun compMacro

[formatUnabbreviated p??]
[sayBrightly p??]
[put p??]
[macroExpand p168]

[$macroIfTrue p??]
[$NoValueMode p165]
[$EmptyMode p166]

— defun compMacro —

```
(defun |compMacro| (form mode env)
 (let (|$macroIfTrue| lhs signature specialCases rhs prhs)
 (declare (special |$macroIfTrue| |$NoValueMode| |$EmptyMode|))
  (setq |$macroIfTrue| t)
  (setq lhs (second form))
  (setq signature (third form))
  (setq specialCases (fourth form))
  (setq rhs (fifth form))
  (setq prhs
   (cond
    ((and (consp rhs) (eq (qfirst rhs) 'category))
      (list "-- the constructor category"))
    ((and (consp rhs) (eq (qfirst rhs) '|Join|))
      (list "-- the constructor category"))
    ((and (consp rhs) (eq (qfirst rhs) 'capsule))
      (list "-- the constructor capsule"))
    ((and (consp rhs) (eq (qfirst rhs) '|add|))
      (list "-- the constructor capsule"))
    (t (|formatUnabbreviated| rhs))))
  (|sayBrightly|
   (cons "   processing macro definition "
     (append (|formatUnabbreviated| lhs)
      (cons " ==> "
       (append prhs (list " "))))))
  (when (or (equal mode |$EmptyMode|) (equal mode |$NoValueMode|))
    (list '|/throwAway| |$NoValueMode|
     (|put| (CAR lhs) '|macro| (|macroExpand| rhs env) env)))))
```

———

6.5.82 defplist compPretend plist

We set up the `compPretend` function to handle the `pretend` keyword by setting the `special` keyword on the `pretend` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|pretend| 'special) '|compPretend|))
```

———

6.5.83 defun compPretend

[addDomain p233]
[comp p530]
[opOf p??]

[stackSemanticError p??]
[stackWarning p??]
[$newCompilerUnionFlag p??]
[$EmptyMode p166]

— defun compPretend —

```
(defun |compPretend| (form mode env)
 (let (x tt warningMessage td tp)
 (declare (special |$newCompilerUnionFlag| |$EmptyMode|))
  (setq x (second form))
  (setq tt (third form))
  (setq env (|addDomain| tt env))
  (when (setq td (or (|comp| x tt env) (|comp| x |$EmptyMode| env)))
   (when (equal (second td) tt)
    (setq warningMessage (list '|pretend| tt '| -- should replace by @|)))
   (cond
     ((and |$newCompilerUnionFlag|
           (eq (|opOf| (second td)) '|Union|)
           (not (eq (|opOf| mode) '|Union|)))
       (|stackSemanticError|
        (list '|cannot pretend | x '| of mode | (second td) '| to mode | mode)
         nil))
     (t
      (setq td (list (first td) tt (third td)))
      (when (setq tp (|coerce| td mode))
        (when warningMessage (|stackWarning| warningMessage))
        tp))))))
```

———-

6.5.84 defplist compQuote plist

We set up the compQuote function to handle the QUOTE keyword by setting the special keyword on the QUOTE symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get 'quote 'special) '|compQuote|))
```

———-

6.5.85 defun compQuote

— defun compQuote —

```
(defun |compQuote| (form mode env)
 (list form mode env))
```

———-

6.5.86 defplist compReduce plist

We set up the compReduce function to handle the REDUCE keyword by setting the special keyword on the REDUCE symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get 'reduce 'special) '|compReduce|))
```

———-

6.5.87 defun compReduce

[compReduce1 p303]
[$formalArgList p??]

— defun compReduce —

```
(defun |compReduce| (form mode env)
  (declare (special |$formalArgList|))
  (|compReduce1| form mode env |$formalArgList|))
```

———-

6.5.88 defun compReduce1

[systemError p??]
[nreverse0 p??]
[compIterator p??]
[comp p530]
[parseTran p99]
[getIdentity p??]
[$sideEffectsList p??]
[$until p??]
[$initList p??]
[$Boolean p??]
[$e p??]
[$endTestList p??]

— defun compReduce1 —

```
(defun |compReduce1| (form mode env |$formalArgList|)
 (declare (special |$formalArgList|))
 (let (|$sideEffectsList| |$until| |$initList| |$endTestList| collectForm
       collectOp body op itl acc afterFirst bodyVal part1 part2 part3 id
       identityCode untilCode finalCode tmp1 tmp2)
 (declare (special |$sideEffectsList| |$until| |$initList| |$Boolean| |$e|
                   |$endTestList|))
  (setq op (second form))
  (setq collectForm (fourth form))
  (setq collectOp (first collectForm))
```

```
(setq tmp1 (reverse (cdr collectForm)))
(setq body (first tmp1))
(setq itl (nreverse (cdr tmp1)))
(when (stringp op) (setq op (intern op)))
(cond
 ((null (member collectOp '(collect collectv collectvec)))
  (|systemError| (list '|illegal reduction form:| form)))
 (t
  (setq |$sideEffectsList| nil)
  (setq |$until| nil)
  (setq |$initList| nil)
  (setq |$endTestList| nil)
  (setq |$e| env)
  (setq itl
   (dolist (x itl (nreverse0 tmp2))
    (setq tmp1 (or (|compIterator| x |$e|) (return '|failed|)))
    (setq |$e| (second tmp1))
    (push (elt tmp1 0) tmp2)))
  (unless (eq itl '|failed|)
   (setq env |$e|)
   (setq acc (gensym))
   (setq afterFirst (gensym))
   (setq bodyVal (gensym))
   (when (setq tmp1 (|comp| (list 'let bodyVal body ) mode env))
    (setq part1 (first tmp1))
    (setq mode (second tmp1))
    (setq env (third tmp1))
    (when (setq tmp1 (|comp| (list 'let acc bodyVal) mode env))
     (setq part2 (first tmp1))
     (setq env (third tmp1))
     (when (setq tmp1
            (|comp| (list 'let acc (|parseTran| (list op acc bodyVal)))
                    mode env))
      (setq part3 (first tmp1))
      (setq env (third tmp1))
      (when (setq identityCode
             (if (setq id (|getIdentity| op env))
              (car (|comp| id mode env))
              (list '|IdentityError| (mkq op))))
       (setq finalCode
        (cons 'progn
         (cons (list 'let afterFirst nil)
          (cons
           (cons 'repeat
            (append itl
             (list
              (list 'progn part1
               (list 'if afterFirst part3
                (list 'progn part2 (list 'let afterFirst (mkq t)))) nil))))
                 (list (list 'if afterFirst acc identityCode ))))))
       (when |$until|
        (setq tmp1 (|comp| |$until| |$Boolean| env))
        (setq untilCode (first tmp1))
        (setq env (third tmp1))
```

```
          (setq finalCode
           (subst (list 'until untilCode) '|$until| finalCode :test #'equal))
          (list finalCode mode env ))))))))))
```

———-

6.5.89 defplist compRepeatOrCollect plist

We set up the compRepeatOrCollect function to handle the COLLECT keyword by setting the special keyword on the COLLECT symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get 'collect 'special) '|compRepeatOrCollect|))
```

———-

6.5.90 defplist compRepeatOrCollect plist

We set up the compRepeatOrCollect function to handle the REPEAT keyword by setting the special keyword on the REPEAT symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get 'repeat 'special) '|compRepeatOrCollect|))
```

———-

6.5.91 defun compRepeatOrCollect

[length p??]
[compIterator p??]
[modeIsAggregateOf p??]
[stackMessage p??]
[compOrCroak p528]
[comp p530]
[coerceExit p330]
[p??]
[p??]
[$until p??]
[$Boolean p??]
[$NoValueMode p165]
[$exitModeStack p??]
[$leaveLevelStack p??]
[$formalArgList p??]

— defun compRepeatOrCollect —

```
(defun |compRepeatOrCollect| (form mode env)
 (labels (
```

```
(fn (form |$exitModeStack| |$leaveLevelStack| |$formalArgList| env)
(declare (special |$exitModeStack| |$leaveLevelStack| |$formalArgList|))
(let (|$until| body itl xp targetMode repeatOrCollect bodyMode bodyp mp tmp1
       untilCode ep itlp formp u mpp tmp2)
(declare (special  |$Boolean| |$until| |$NoValueMode| ))
 (setq |$until| nil)
 (setq repeatOrCollect (car form))
 (setq tmp1 (reverse (cdr form)))
 (setq body (car tmp1))
 (setq itl (nreverse (cdr tmp1)))
 (setq itlp
  (dolist (x itl (nreverse0 tmp2))
   (setq tmp1 (or (|compIterator| x env) (return '|failed|)))
   (setq xp (first tmp1))
   (setq env (second tmp1))
   (push xp tmp2)))
 (unless (eq itlp '|failed|)
  (setq targetMode (car |$exitModeStack|))
  (setq bodyMode
    (if (eq repeatOrCollect 'collect)
     (cond
      ((eq targetMode '|$EmptyMode|)
        '|$EmptyMode|)
      ((setq u (|modeIsAggregateOf| '|List| targetMode env))
        (second u))
      ((setq u (|modeIsAggregateOf| '|PrimitiveArray| targetMode env))
       (setq repeatOrCollect 'collectv)
       (second u))
      ((setq u (|modeIsAggregateOf| '|Vector| targetMode env))
       (setq repeatOrCollect 'collectvec)
       (second u))
      (t
       (|stackMessage| "Invalid collect bodytype")
       '|failed|))
      |$NoValueMode|))
   (unless (eq bodyMode '|failed|)
    (when (setq tmp1 (|compOrCroak| body bodyMode env))
     (setq bodyp (first tmp1))
     (setq mp (second tmp1))
     (setq ep (third tmp1))
     (when |$until|
       (setq tmp1 (|comp| |$until| |$Boolean| ep))
       (setq untilCode (first tmp1))
       (setq ep (third tmp1))
       (setq itlp
         (subst (list 'until untilCode) '|$until| itlp :test #'equal)))
     (setq formp (cons repeatOrCollect (append itlp (list bodyp))))
     (setq mpp
      (cond
       ((eq repeatOrCollect 'collect)
         (if (setq u (|modeIsAggregateOf| '|List| targetMode env))
           (car u)
           (list '|List| mp)))
       ((eq repeatOrCollect 'collectv)
```

```
          (if (setq u (|modeIsAggregateOf| '|PrimitiveArray| targetMode env))
            (car u)
            (list '|PrimitiveArray| mp)))
        ((eq repeatOrCollect 'collectvec)
          (if (setq u (|modeIsAggregateOf| '|Vector| targetMode env))
            (car u)
            (list '|Vector| mp)))
        (t mp)))
      (|coerceExit| (list formp  mpp ep) targetMode)))))) )
 (declare (special |$exitModeStack| |$leaveLevelStack| |$formalArgList|))
 (fn form
   (cons mode |$exitModeStack|)
   (cons (|#| |$exitModeStack|) |$leaveLevelStack|)
   |$formalArgList|
   env)))
```

———-

6.5.92 defplist compReturn plist

We set up the `compReturn` function to handle the `return` keyword by setting the `special` keyword on the `return` symbol property list.

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|return| 'special) '|compReturn|))
```

———-

6.5.93 defun compReturn

[stackSemanticError p??]
[userError p??]
[resolve p334]
[comp p530]
[modifyModeStack p561]
[$exitModeStack p??]
[$returnMode p??]

— **defun compReturn** —

```
(defun |compReturn| (form mode env)
 (let (level x index u xp mp ep)
 (declare (special |$returnMode| |$exitModeStack|))
  (setq level (second form))
  (setq x (third form))
  (cond
   ((null |$exitModeStack|)
    (|stackSemanticError|
      (list '|the return before| x '|is unneccessary|) nil)
    nil)
```

```
   ((not (eql level 1))
     (|userError| "multi-level returns not supported"))
   (t
    (setq index (max 0 (1- (|#| |$exitModeStack|))))
    (when (>= index 0)
      (setq |$returnMode|
       (|resolve| (elt |$exitModeStack| index) |$returnMode|)))
    (when (setq u (|comp| x |$returnMode| env))
     (setq xp (first u))
     (setq mp (second u))
     (setq ep (third u))
     (when (>= index 0)
       (setq |$returnMode| (|resolve| mp |$returnMode|))
       (|modifyModeStack| mp index))
     (list (list '|TAGGEDreturn| 0 u) mode ep))))))
```

———-

6.5.94 defplist compSeq plist

We set up the `compSeq` function to handle the `SEQ` keyword by setting the `special` keyword on the `SEQ` symbol property list.

— **postvars** —

```
(eval-when (eval load)
 (setf (get 'seq 'special) '|compSeq|))
```

———-

6.5.95 defun compSeq

[compSeq1 p308]
[$exitModeStack p??]

— **defun compSeq** —

```
(defun |compSeq| (form mode env)
 (declare (special |$exitModeStack|))
 (|compSeq1| (cdr form) (cons mode |$exitModeStack|) env))
```

———-

6.5.96 defun compSeq1

[nreverse0 p??]
[compSeqItem p310]
[mkq p??]
[replaceExitEtc p309]
[$exitModeStack p??]
[$insideExpressionIfTrue p??]

[$finalEnv p??]
[$NoValueMode p165]

— defun compSeq1 —

```
(defun |compSeq1| (form |$exitModeStack| env)
 (declare (special |$exitModeStack|))
 (let (|$insideExpressionIfTrue| |$finalEnv| tmp1 tmp2 c catchTag newform)
 (declare (special |$insideExpressionIfTrue| |$finalEnv| |$NoValueMode|))
  (setq |$insideExpressionIfTrue| nil)
  (setq |$finalEnv| nil)
  (when
    (setq c (dolist (x form (nreverse0 tmp2))
             (setq |$insideExpressionIfTrue| nil)
             (setq tmp1 (|compSeqItem| x |$NoValueMode| env))
             (unless tmp1 (return nil))
             (setq env (third tmp1))
             (push (first tmp1) tmp2)))
   (setq catchTag (mkq (gensym)))
   (setq newform
    (cons 'seq
     (|replaceExitEtc| c catchTag '|TAGGEDexit| (elt |$exitModeStack| 0))))
   (list (list 'catch catchTag newform)
         (elt |$exitModeStack| 0) |$finalEnv|))))
```

———-

6.5.97 defun replaceExitEtc

[rplac p??]
[replaceExitEtc p309]
[intersectionEnvironment p??]
[convertOrCroak p310]
[$finalEnv p??]
[$finalEnv p??]

— defun replaceExitEtc —

```
(defun |replaceExitEtc| (x tag opFlag opMode)
 (declare (special |$finalEnv|))
  (cond
   ((atom x) nil)
   ((and (consp x) (eq (qfirst x) 'quote)) nil)
   ((and (consp x) (equal (qfirst x) opFlag) (consp (qrest x))
         (consp (qcddr x)) (eq (qcdddr x) nil))
     (|rplac| (caaddr x) (|replaceExitEtc| (caaddr x) tag opFlag opMode))
     (cond
      ((eql (second x) 0)
        (setq |$finalEnv|
         (if |$finalEnv|
           (|intersectionEnvironment| |$finalEnv| (third (third x)))
           (third (third x))))
         (|rplac| (car x) 'throw)
```

```
         (|rplac| (cadr x) tag)
         (|rplac| (caddr x) (car (|convertOrCroak| (caddr x) opMode))))
      (t
        (|rplac| (cadr x) (1- (cadr x))))))
   ((and (consp x) (consp (qrest x)) (consp (qcddr x))
         (eq (qcdddr x) nil)
         (member (qfirst x) '(|TAGGEDreturn| |TAGGEDexit|)))
     (|rplac| (car (caddr x))
       (|replaceExitEtc| (car (caddr x)) tag opFlag opMode)))
   (t
     (|replaceExitEtc| (car x) tag opFlag opMode)
     (|replaceExitEtc| (cdr x) tag opFlag opMode)))
 x)
```

———-

6.5.98 defun convertOrCroak

[convert p538]
[userError p??]

— defun convertOrCroak —

```
(defun |convertOrCroak| (tt m)
 (let (u)
 (if (setq u (|convert| tt m))
   u
   (|userError|
    (list '|CANNOT CONVERT: | (first tt) '|%l| '| OF MODE: | (second tt)
          '|%l| '| TO MODE: | m  '|%l|)))))
```

———-

6.5.99 defun compSeqItem

[comp p530]
[macroExpand p168]

— defun compSeqItem —

```
(defun |compSeqItem| (form mode env)
  (|comp| (|macroExpand| form env) mode env))
```

———-

6.5.100 defplist compSetq plist

We set up the `compSetq` function to handle the `LET` keyword by setting the `special` keyword on the `LET` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get 'let 'special) '|compSetq|))
```

———-

6.5.101 defplist compSetq plist

We set up the `compSetq` function to handle the `SETQ` keyword by setting the `special` keyword on the `SETQ` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get 'setq 'special) '|compSetq|))
```

———-

6.5.102 defun compSetq

[compSetq1 p311]

— defun compSetq —

```
(defun |compSetq| (form mode env)
 (|compSetq1| (second form) (third form) mode env))
```

———-

6.5.103 defun compSetq1

[setqSingle p315]
[compSetq1 identp (vol5)]
[compMakeDeclaration p561]
[compSetq p311]
[setqMultiple p312]
[setqSetelt p315]
[$EmptyMode p166]

— defun compSetq1 —

```
(defun |compSetq1| (form val mode env)
 (let (x y ep op z)
 (declare (special |$EmptyMode|))
 (cond
  ((identp form) (|setqSingle| form val mode env))
  ((and (consp form) (eq (qfirst form) '|:|) (consp (qrest form))
        (consp (qcddr form)) (eq (qcdddr form) nil))
    (setq x (second form))
    (setq y (third form))
    (setq ep (third (|compMakeDeclaration| form |$EmptyMode| env)))
    (|compSetq| (list 'let x val) mode ep))
```

```
  ((consp form)
    (setq op (qfirst form))
    (setq z (qrest form))
    (cond
     ((eq op 'cons)      (|setqMultiple| (|uncons| form) val mode env))
     ((eq op '|@Tuple|) (|setqMultiple| z val mode env))
     (t                  (|setqSetelt| form val mode env)))))))
```

———-

6.5.104 defun uncons

[uncons p312]

— defun uncons —

```
(defun |uncons| (x)
  (cond
   ((atom x) x)
   ((and (consp x) (eq (qfirst x) 'cons)  (consp (qrest x))
         (consp (qcddr x)) (eq (qcdddr x) nil))
      (cons (second x) (|uncons| (third x))))))
```

———-

6.5.105 defun setqMultiple

[nreverse0 p??]
[stackMessage p??]
[setqMultipleExplicit p314]
[genVariable p??]
[addBinding p??]
[compSetq1 p311]
[convert p538]
[put p??]
[genSomeVariable p??]
[length p??]
[mkprogn p??]
[$EmptyMode p166]
[$NoValueMode p165]
[$noEnv p??]

— defun setqMultiple —

```
(defun |setqMultiple| (nameList val m env)
 (labels (
  (decompose (tt len env)
   (declare (ignore len))
   (let (tmp1 z)
   (declare (special |$EmptyMode|))
    (cond
```

```
     ((and (consp tt) (eq (qfirst tt) '|Record|)
           (progn (setq z (qrest tt)) t))
      (loop for item in z
       collect (cons (second item) (third item))))
     ((progn
       (setq tmp1 (|comp| tt |$EmptyMode| env))
       (and (consp tmp1) (consp (qrest tmp1)) (consp (qsecond tmp1))
            (eq (qcaadr tmp1) '|RecordCategory|)
            (consp (qcddr tmp1)) (eq (qcdddr tmp1) nil)))
      (loop for item in z
       collect (cons (second item) (third item))))
     (t (|stackMessage| (list '|no multiple assigns to mode: | tt)))))))
 (let (g m1 tt x mp selectorModePairs tmp2 assignList)
 (declare (special |$noEnv| |$EmptyMode| |$NoValueMode|))
  (cond
   ((and (consp val) (eq (qfirst val) 'cons) (equal m |$NoValueMode|))
     (|setqMultipleExplicit| nameList (|uncons| val) m env))
   ((and (consp val) (eq (qfirst val) '|@Tuple|) (equal m |$NoValueMode|))
     (|setqMultipleExplicit| nameList (qrest val) m env))
   ; 1 create a gensym, %add to local environment, compile and assign rhs
   (t
     (setq g (|genVariable|))
     (setq env (|addBinding| g nil env))
     (setq tmp2 (|compSetq1| g val |$EmptyMode| env))
     (when tmp2
      (setq tt tmp2)
      (setq m1 (cadr tmp2))
      (setq env (|put| g 'mode m1 env))
      (setq tmp2 (|convert| tt m))
;  1.1 --exit if result is a list
      (when tmp2
       (setq x (first tmp2))
       (setq mp (second tmp2))
       (setq env (third tmp2))
       (cond
        ((and (consp m1) (eq (qfirst m1) '|List|) (consp (qrest m1))
              (eq (qcddr m1) nil))
         (loop for y in nameList do
          (setq env
           (|put| y '|value| (list (|genSomeVariable|) (second m1) |$noEnv|)
           env)))
         (|convert| (list (list 'progn x (list 'let nameList g) g) mp env) m))
        (t
;  2 --verify that the #nameList = number of parts of right-hand-side
         (setq selectorModePairs
          (decompose m1 (|#| nameList) env))
         (when selectorModePairs
          (cond
           ((not (eql (|#| nameList) (|#| selectorModePairs)))
             (|stackMessage|
              (list val '| must decompose into |
               (|#| nameList) '| components| )))
           (t
;  3 --generate code
```

```
          (setq assignList
           (loop for x in nameList
                 for item in selectorModePairs
            collect (car
                     (progn
                      (setq tmp2
                        (or (|compSetq1| x (list '|elt| g (first item))
                                           (rest item) env)
                            (return '|failed|)))
                      (setq env (third tmp2))
                      tmp2))))
          (unless (eq assignList '|failed|)
           (list (mkprogn (cons x (append assignList (list g)))) mp env))
            ))))))))))
```

———-

6.5.106 defun setqMultipleExplicit

[stackMessage p??]
[genVariable p??]
[compSetq1 p311]
[last p??]
[$EmptyMode p166]
[$NoValueMode p165]

— defun setqMultipleExplicit —

```
(defun |setqMultipleExplicit| (nameList valList m env)
 (declare (ignore m))
 (let (gensymList assignList tmp1 reAssignList)
 (declare (special |$NoValueMode| |$EmptyMode|))
  (cond
   ((not (eql (|#| nameList) (|#| valList)))
    (|stackMessage|
     (list '|Multiple assignment error; # of items in: | nameList
           '|must = # in: | valList)))
   (t
    (setq gensymList
     (loop for name in nameList
      collect (|genVariable|)))
    (setq assignList
     (loop for g in gensymList
           for val in valList
      collect (progn
                (setq tmp1
                 (or (|compSetq1| g val |$EmptyMode| env)
                     (return '|failed|)))
                (setq env (third tmp1))
                tmp1)))
    (unless (eq assignList '|failed|)
     (setq reAssignList
```

```
      (loop for g in gensymList
            for name in nameList
       collect (progn
                (setq tmp1
                 (or (|compSetq1| name g |$EmptyMode| env)
                     (return '|failed|)))
                 (setq env (third tmp1))
                 tmp1)))
     (unless (eq reAssignList '|failed|)
      (list
       (cons 'progn
        (append
         (loop for tt in assignList
          collect (car tt))
         (loop for tt in reAssignList
          collect (car tt))))
         |$NoValueMode| (third (|last| reAssignList))))))))
```

———-

6.5.107 defun setqSetelt

[comp p530]

— defun setqSetelt —

```
(defun |setqSetelt| (form val mode env)
 (|comp| (cons '|setelt| (cons (car form) (append (cdr form) (list val))))
          mode env))
```

———-

6.5.108 defun setqSingle

[setqSingle getProplist (vol5)]
[getmode p??]
[get p??]
[maxSuperType p318]
[comp p530]
[getmode p??]
[assignError p317]
[convert p538]
[setqSingle identp (vol5)]
[profileRecord p??]
[consProplistOf p??]
[removeEnv p??]
[setqSingle addBinding (vol5)]
[isDomainForm p319]
[isDomainInScope p??]
[stackWarning p??]

[augModemapsFromDomain1 p237]
[NRTassocIndex p317]
[isDomainForm p319]
[outputComp p318]
[$insideSetqSingleIfTrue p??]
[$QuickLet p??]
[$form p??]
[$profileCompiler p??]
[$EmptyMode p166]
[$NoValueMode p165]

— defun setqSingle —

```
(defun |setqSingle| (form val mode env)
 (let (|$insideSetqSingleIfTrue| currentProplist mpp maxmpp td x mp tp key
       newProplist ep k newform)
 (declare (special |$insideSetqSingleIfTrue| |$QuickLet| |$form|
                   |$profileCompiler| |$EmptyMode| |$NoValueMode|))
  (setq |$insideSetqSingleIfTrue| t)
  (setq currentProplist (|getProplist| form env))
  (setq mpp
   (or (|get| form '|mode| env) (|getmode| form env)
       (if (equal mode |$NoValueMode|) |$EmptyMode| mode)))
  (when (setq td
         (cond
          ((setq td (|comp| val mpp env))
           td)
          ((and (null (|get| form '|mode| env))
                (not (equal mpp (setq maxmpp (|maxSuperType| mpp env))))
                (setq td (|comp| val maxmpp env)))
           td)
          ((and (setq td (|comp| val |$EmptyMode| env))
                (|getmode| (second td) env))
           (|assignError| val (second td) form mpp))))
   (when (setq tp (|convert| td mode))
    (setq x (first tp))
    (setq mp (second tp))
    (setq ep (third tp))
    (when (and |$profileCompiler| (identp form))
      (setq key (if (member form (cdr |$form|)) '|arguments| '|locals|))
      (|profileRecord| key form (second td)))
    (setq newProplist
     (|consProplistOf| form currentProplist '|value|
      (|removeEnv| (cons val (cdr td)))))
    (setq ep (if (consp form) ep (|addBinding| form newProplist ep)))
    (when (|isDomainForm| val ep)
     (when (|isDomainInScope| form ep)
      (|stackWarning|
       (list '|domain valued variable| form
             '|has been reassigned within its scope| )))
      (setq ep (|augModemapsFromDomain1| form val ep)))
    (if (setq k (|NRTassocIndex| form))
      (setq newform (list 'setelt '$ k x))
      (setq newform
```

```
      (if |$QuickLet|
       (list 'let form x)
       (list 'let form x
         (if (|isDomainForm| x ep)
           (list 'elt form 0)
           (car (|outputComp| form ep)))))))
    (list newform mp ep)))))
```

——————

6.5.109 defun NRTassocIndex

This function returns the index of domain entry x in the association list [$NRTaddForm p??]
[$NRTdeltaList p??]
[$found p??]
[$NRTbase p??]
[$NRTdeltaLength p??]

— defun NRTassocIndex —

```
(defun |NRTassocIndex| (x)
 (let (k (i 0))
 (declare (special |$NRTdeltaLength| |$NRTbase| |$found| |$NRTdeltaList|
                   |$NRTaddForm|))
 (cond
  ((null x) x)
  ((equal x |$NRTaddForm|) 5)
  ((setq k
    (let (result)
     (loop for y in |$NRTdeltaList|
      when (and (incf i)
                (eq (elt y 0) '|domain|)
                (equal (elt y 1) x)
                (setq |$found| y))
      do (setq result (or result i)))
     result))
   (- (+ |$NRTbase| |$NRTdeltaLength|) k))
  (t nil))))
```

——————

6.5.110 defun assignError

[stackMessage p??]

— defun assignError —

```
(defun |assignError| (val mp form m)
 (let (message)
  (setq message
   (if val
    (list '|CANNOT ASSIGN: | val '|%l|
```

```
        '|   OF MODE: | mp '|%l|
        '|   TO: | form '|%l| '|   OF MODE: | m)
    (list '|CANNOT ASSIGN: | val '|%l|
        '|   TO: | form '|%l| '|   OF MODE: | m)))
  (|stackMessage| message)))
```

———-

6.5.111 defun outputComp

[comp p530]
[nreverse0 p??]
[outputComp p318]
[get p??]
[$Expression p??]

— defun outputComp —

```
(defun |outputComp| (x env)
 (let (argl v)
 (declare (special |$Expression|))
  (cond
   ((|comp| (list '|::| x |$Expression|) |$Expression| env))
   ((and (consp x) (eq (qfirst x) '|construct|))
     (setq argl (qrest x))
     (list (cons 'list
      (let (result tmp1)
       (loop for x in argl
        do (setq result
            (cons (car
             (progn
              (setq tmp1 (|outputComp| x env))
              (setq env (third tmp1))
               tmp1))
              result)))
         (nreverse0 result)))
        |$Expression| env))
     ((and (setq v (|get| x '|value| env))
           (consp (cadr v)) (eq (qfirst (cadr v)) '|Union|))
        (list (list '|coerceUn2E| x (cadr v)) |$Expression| env))
     (t (list x |$Expression| env)))))
```

———-

6.5.112 defun maxSuperType

[get p??]
[maxSuperType p318]

— defun maxSuperType —

```
(defun |maxSuperType| (m env)
 (let (typ)
  (if (setq typ (|get| m '|SuperDomain| env))
   (|maxSuperType| typ env)
   m)))
```

——–

6.5.113 defun isDomainForm

[isFunctor p234]
[isCategoryForm p??]
[isDomainConstructorForm p319]
[$SpecialDomainNames p??]

— defun isDomainForm —

```
(defun |isDomainForm| (d env)
 (let (tmp1)
 (declare (special |$SpecialDomainNames|))
  (or (member (ifcar d) |$SpecialDomainNames|) (|isFunctor| d)
      (and (progn
             (setq tmp1 (|getmode| d env))
             (and (consp tmp1) (eq (qfirst tmp1) '|Mapping|) (consp (qrest tmp1))))
           (|isCategoryForm| (qsecond tmp1) env))
    (|isCategoryForm| (|getmode| d env) env)
    (|isDomainConstructorForm| d env))))
```

——–

6.5.114 defun isDomainConstructorForm

[isCategoryForm p??]
[eqsubstlist p??]
[$FormalMapVariableList p249]

— defun isDomainConstructorForm —

```
(defun |isDomainConstructorForm| (d env)
 (let (u)
 (declare (special |$FormalMapVariableList|))
  (when
   (and (consp d)
        (setq u (|get| (qfirst d) '|value| env))
        (consp u)
        (consp (qrest u))
        (consp (qsecond u))
        (eq (qcaadr u) '|Mapping|)
        (consp (qcdadr u)))
    (|isCategoryForm|
     (eqsubstlist (rest d) |$FormalMapVariableList| (cadadr u)) env))))
```

———-

6.5.115 defplist compString plist

We set up the `compString` function to handle the `String` keyword by setting the `special` keyword on the `String` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|String| 'special) '|compString|))
```

———-

6.5.116 defun compString

[resolve p334]
[$StringCategory p??]

— defun compString —

```
(defun |compString| (form mode env)
 (declare (special |$StringCategory|))
 (list form (|resolve| |$StringCategory| mode) env))
```

———-

6.5.117 defplist compSubDomain plist

We set up the `compSubDomain` function to handle the `SubDomain` keyword by setting the `special` keyword on the `SubDomain` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|SubDomain| 'special) '|compSubDomain|))
```

———-

6.5.118 defun compSubDomain

[compSubDomain1 p321]
[compCapsule p256]
[$addFormLhs p??]
[$NRTaddForm p??]
[$addForm p??]
[$addFormLhs p??]

— defun compSubDomain —

```
(defun |compSubDomain| (form mode env)
 (let (|$addFormLhs| |$addForm| domainForm predicate tmp1)
 (declare (special |$addFormLhs| |$addForm| |$NRTaddForm| |$addFormLhs|))
  (setq domainForm (second form))
  (setq predicate (third form))
  (setq |$addFormLhs| domainForm)
  (setq |$addForm| nil)
  (setq |$NRTaddForm| domainForm)
  (setq tmp1 (|compSubDomain1| domainForm predicate mode env))
  (setq |$addForm| (first tmp1))
  (setq env (third tmp1))
  (|compCapsule| (list 'capsule) mode env)))
```

———-

6.5.119 defun compSubDomain1

[compMakeDeclaration p561]
[addDomain p233]
[compOrCroak p528]
[stackSemanticError p??]
[lispize p322]
[evalAndRwriteLispForm p191]
[$CategoryFrame p??]
[$op p??]
[$lisplibSuperDomain p??]
[$Boolean p??]
[$EmptyMode p166]

— defun compSubDomain1 —

```
(defun |compSubDomain1| (domainForm predicate mode env)
 (let (u prefixPredicate opp dFp)
 (declare (special |$CategoryFrame| |$op| |$lisplibSuperDomain| |$Boolean|
                   |$EmptyMode|))
  (setq env (third
   (|compMakeDeclaration| (list '|:| '|#1| domainForm)
     |$EmptyMode| (|addDomain| domainForm env))))
   (setq u (|compOrCroak| predicate |$Boolean| env))
   (unless u
    (|stackSemanticError|
     (list '|predicate: | predicate
           '| cannot be interpreted with #1: | domainForm) nil))
   (setq prefixPredicate (|lispize| (first u)))
   (setq |$lisplibSuperDomain| (list domainForm predicate))
   (|evalAndRwriteLispForm| '|evalOnLoad2|
    (list 'setq '|$CategoryFrame|
     (list '|put|
      (setq opp (list 'quote |$op|))
       ''|SuperDomain|
        (setq dFp (list 'quote domainForm))
         (list '|put| dFp ''|SubDomain|
```

```
        (list 'cons (list 'quote (cons |$op| prefixPredicate))
        (list 'delasc opp (list '|get| dFp ''|SubDomain| '|$CategoryFrame|)))
        '|$CategoryFrame|))))
  (list domainForm mode env)))
```

———-

6.5.120 defun lispize

[optimize p213]

— defun lispize —

```
(defun |lispize| (x)
 (car (|optimize| (list x))))
```

———-

6.5.121 defplist compSubsetCategory plist

We set up the `compSubsetCategory` function to handle the `SubsetCategory` keyword by setting the `special` keyword on the `SubsetCategory` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|SubsetCategory| 'special) '|compSubsetCategory|))
```

———-

6.5.122 defun compSubsetCategory

TPDHERE: See LocalAlgebra for an example call [put p??]
[comp p530]
[$lhsOfColon p??]

— defun compSubsetCategory —

```
(defun |compSubsetCategory| (form mode env)
 (let (cat r)
 (declare (special |$lhsOfColon|))
  (setq cat (second form))
  (setq r (third form))
  ;  --1. put "Subsets" property on R to allow directly coercion to subset;
  ;  --   allow automatic coercion from subset to R but not vice versa
  (setq env (|put| r '|Subsets| (list (list |$lhsOfColon| '|isFalse|)) env))
  ;  --2. give the subset domain modemaps of cat plus 3 new functions
  (|comp|
    (list '|Join| cat
      (subst |$lhsOfColon| '$
             (list 'category '|domain|
              (list 'signature '|coerce| (list r '$))
```

```
            (list 'signature '|lift| (list r '$))
             (list 'signature '|reduce| (list '$ r))) :test #'equal))
     mode env)))
```

———-

6.5.123 defplist compSuchthat plist

We set up the compSuchthat function to handle the | keyword by setting the special keyword on the | symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '\| 'special) '|compSuchthat|))
```

———-

6.5.124 defun compSuchthat

[comp p530]
[put p??]
[$Boolean p??]

— defun compSuchthat —

```
(defun |compSuchthat| (form mode env)
 (let (x p xp mp tmp1 pp)
 (declare (special |$Boolean|))
  (setq x (second form))
  (setq p (third form))
  (when (setq tmp1 (|comp| x mode env))
   (setq xp (first tmp1))
   (setq mp (second tmp1))
   (setq env (third tmp1))
   (when (setq tmp1 (|comp| p |$Boolean| env))
    (setq pp (first tmp1))
    (setq env (third tmp1))
    (setq env (|put| xp '|condition| pp env))
    (list xp mp env)))))
```

———-

6.5.125 defplist compVector plist

We set up the compVector function to handle the VECTOR keyword by setting the special keyword on the VECTOR symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get 'vector 'special) '|compVector|))
```

———-

6.5.126 defun compVector

```
;  null l => [$EmptyVector,m,e]
;  Tl:= [[.,mUnder,e]:= comp(x,mUnder,e) or return "failed" for x in l]
;  Tl="failed" => nil
;  [["VECTOR",:[T.expr for T in Tl]],m,e]
```

[comp p530]
[$EmptyVector p??]

— defun compVector —

```
(defun |compVector| (form mode env)
 (let (tmp1 tmp2 t0 failed (newmode (second mode)))
  (declare (special |$EmptyVector|))
   (if (null form)
    (list |$EmptyVector| mode env)
    (progn
     (setq t0
      (do ((t3 form (cdr t3)) (x nil))
          ((or (atom t3) failed) (unless failed (nreverse0 tmp2)))
        (setq x (car t3))
        (if (setq tmp1 (|comp| x newmode env))
          (progn
           (setq newmode (second tmp1))
           (setq env (third tmp1))
           (push tmp1 tmp2))
          (setq failed t))))
     (unless failed
      (list (cons 'vector
                  (loop for texpr in t0 collect (car texpr))) mode env))))))
```

———-

6.5.127 defplist compWhere plist

We set up the `compWhere` function to handle the `where` keyword by setting the `special` keyword on the `where` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|where| 'special) '|compWhere|))
```

———-

6.5.128 defun compWhere

[comp p530]
[macroExpand p168]

[deltaContour p??]
[addContour p??]
[$insideExpressionIfTrue p??]
[$insideWhereIfTrue p??]
[$EmptyMode p166]

— defun compWhere —

```
(defun |compWhere| (form mode eInit)
 (let (|$insideExpressionIfTrue| |$insideWhereIfTrue| newform exprList e
       eBefore tmp1 x eAfter del eFinal)
 (declare (special |$insideExpressionIfTrue| |$insideWhereIfTrue|
                   |$EmptyMode|))
  (setq newform (second form))
  (setq exprlist (cddr form))
  (setq |$insideExpressionIfTrue| nil)
  (setq |$insideWhereIfTrue| t)
  (setq e eInit)
  (when (dolist (item exprList t)
          (setq tmp1 (|comp| item |$EmptyMode| e))
          (unless tmp1 (return nil))
          (setq e (third tmp1)))
  (setq |$insideWhereIfTrue| nil)
  (setq tmp1 (|comp| (|macroExpand| newform (setq eBefore e)) mode e))
  (when tmp1
    (setq x (first tmp1))
    (setq mode (second tmp1))
    (setq eAfter (third tmp1))
    (setq del (|deltaContour| eAfter eBefore))
    (if del
      (setq eFinal (|addContour| del eInit))
      (setq eFinal eInit))
    (list x mode eFinal)))))
```

———–

6.6 Functions for coercion

6.6.1 defun coerce

The function coerce is used by the old compiler for coercions. The function coerceInteractive is used by the interpreter. One should always call the correct function, since the representation of basic objects may not be the same. [keyedSystemError p??]
[rplac p??]
[coerceEasy p326]
[coerceSubset p327]
[coerceHard p327]
[isSomeDomainVariable p??]
[stackMessage p??]
[$InteractiveMode p??]
[$Rep p??]

[$fromCoerceable p??]

— defun coerce —

```
(defun |coerce| (tt mode)
 (labels (
  (fn (x m1 m2)
   (list '|Cannot coerce| x '|%l| '|      of mode| m1
         '|%l| '|      to mode| m2)))
 (let (tp)
 (declare (special |$fromCoerceable$| |$Rep| |$InteractiveMode|))
  (if |$InteractiveMode|
   (|keyedSystemError|
      "Unexpected error or improper call to system function %1: %2"
    (list "coerce" "function coerce called from the interpreter."))
   (progn
    (|rplac| (cadr tt) (subst '$ |$Rep| (cadr tt) :test #'equal))
    (cond
     ((setq tp (|coerceEasy| tt mode)) tp)
     ((setq tp (|coerceSubset| tt mode)) tp)
     ((setq tp (|coerceHard| tt mode)) tp)
     ((or (eq (car tt) '|$fromCoerceable$|) (|isSomeDomainVariable| mode)) nil)
     (t (|stackMessage| (fn (first tt) (second tt) mode)))))))))
```

———-

6.6.2 defun coerceEasy

[modeEqualSubst p336]
[$EmptyMode p166]
[$Exit p??]
[$NoValueMode p165]
[$Void p??]

— defun coerceEasy —

```
(defun |coerceEasy| (tt m)
  (declare (special |$EmptyMode| |$Exit| |$NoValueMode| |$Void|))
  (cond
    ((equal m |$EmptyMode|) tt)
    ((or (equal m |$NoValueMode|) (equal m |$Void|))
     (list (car tt) m (third tt)))
    ((equal (second tt) m) tt)
    ((equal (second tt) |$NoValueMode|) tt)
    ((equal (second tt) |$Exit|)
     (list
       (list 'progn (car tt) (list '|userError| "Did not really exit."))
       m (third tt)))
    ((or (equal (second tt) |$EmptyMode|)
         (|modeEqualSubst| (second tt) m (third tt)))
     (list (car tt) m (third tt)))))
```

———-

6.6.3 defun coerceSubset

[isSubset p??]
[lassoc p??]
[get p??]
[opOf p??]
[eval p??]
[isSubset p??]
[maxSuperType p318]

— defun coerceSubset —

```
(defun |coerceSubset| (arg1 mp)
 (let (x m env pred)
  (setq x (first arg1))
  (setq m (second arg1))
  (setq env (third arg1))
  (cond
   ((or (|isSubset| m mp env) (and (eq m '|Rep|) (eq mp '$)))
     (list x mp env))
   ((and (consp m) (eq (qfirst m) '|SubDomain|)
         (consp (qrest m)) (equal (qsecond m) mp))
     (list x mp env))
   ((and (setq pred (lassoc (|opOf| mp) (|get| (|opOf| m) '|SubDomain| env)))
          (integerp x) (|eval| (subst x '|#1| pred :test #'equal)))
     (list x mp env))
   ((and (setq pred (|isSubset| mp (|maxSuperType| m env) env))
          (integerp x) (|eval| (subst x '* pred :test #'equal)))
     (list x mp env))
   (t nil))))
```

———-

6.6.4 defun coerceHard

[modeEqual p335]
[get p??]
[getmode p??]
[isCategoryForm p??]
[extendsCategoryForm p??]
[coerceExtraHard p328]
[$e p??]
[$e p??]
[$String p320]
[$bootStrapMode p??]

— defun coerceHard —

```
(defun |coerceHard| (tt m)
 (let (|$e| mp tmp1 mpp)
 (declare (special |$e| |$String| |$bootStrapMode|))
  (setq |$e| (third tt))
```

```
(setq mp (second tt))
(cond
 ((and (stringp mp) (|modeEqual| m |$String|))
   (list (car tt) m  |$e|))
 ((or (|modeEqual| mp m)
      (and (or (progn
                (setq tmp1 (|get| mp '|value| |$e|))
                (and (consp tmp1)
                (progn (setq mpp (qfirst tmp1)) t)))
               (progn
                 (setq tmp1 (|getmode| mp |$e|))
                 (and (consp tmp1)
                      (eq (qfirst tmp1) '|Mapping|)
                      (and (consp (qrest tmp1))
                           (eq (qcddr tmp1) nil)
                           (progn (setq mpp (qsecond tmp1)) t)))))
            (|modeEqual| mpp m))
      (and (or (progn
                (setq tmp1 (|get| m '|value| |$e|))
                (and (consp tmp1)
                (progn (setq mpp (qfirst tmp1)) t)))
               (progn
                (setq tmp1 (|getmode| m |$e|))
                (and (consp tmp1)
                     (eq (qfirst tmp1) '|Mapping|)
                     (and (consp (qrest tmp1))
                          (eq (qcddr tmp1) nil)
                          (progn (setq mpp (qsecond tmp1)) t)))))
             (|modeEqual| mpp mp)))
   (list (car tt) m (third tt)))
 ((and (stringp (car tt)) (equal (car tt) m))
   (list (car tt) m |$e|))
 ((|isCategoryForm| m |$e|)
   (cond
    ((eq |$bootStrapMode| t)
      (list (car tt) m |$e|))
    ((|extendsCategoryForm| (car tt) (cadr tt) m)
      (list (car tt) m |$e|))
    (t (|coerceExtraHard| tt m))))
 (t (|coerceExtraHard| tt m)))))
```

———-

6.6.5 defun coerceExtraHard

[autoCoerceByModemap p333]
[isUnionMode p295]
[hasType p329]
[member p??]
[autoCoerceByModemap p333]
[coerce p325]
[$Expression p??]

— defun coerceExtraHard —

```
(defun |coerceExtraHard| (tt m)
 (let (x mp e tmp1 z ta tp tpp)
 (declare (special |$Expression|))
  (setq x (first tt))
  (setq mp (second tt))
  (setq e (third tt))
  (cond
   ((setq tp (|autoCoerceByModemap| tt m)) tp)
   ((and (progn
          (setq tmp1 (|isUnionMode| mp e))
          (and (consp tmp1) (eq (qfirst tmp1) '|Union|)
          (progn
           (setq z (qrest tmp1)) t)))
           (setq ta (|hasType| x e))
           (|member| ta z)
           (setq tp (|autoCoerceByModemap| tt ta))
           (setq tpp (|coerce| tp m)))
       tpp)
   ((and (consp mp) (eq (qfirst mp) '|Record|) (equal m |$Expression|))
     (list (list '|coerceRe2E| x (list 'elt (copy mp) 0)) m e))
   (t nil))))
```

———-

6.6.6 defun hasType

[get p**??**]

— defun hasType —

```
(defun |hasType| (x e)
 (labels (
  (fn (x)
   (cond
    ((null x) nil)
    ((and (consp x) (consp (qfirst x)) (eq (qcaar x) '|case|)
          (consp (qcdar x)) (consp (qcddar x))
          (eq (qcdddar x) nil))
       (qcaddar x))
    (t (fn (cdr x))))))
  (fn (|get| x '|condition| e))))
```

———-

6.6.7 defun coerceable

[pmatch p**??**]
[sublis p**??**]
[coerce p325]

[$fromCoerceable p??]

— defun coerceable —

```
(defun |coerceable| (m mp env)
 (let (sl)
 (declare (special |$fromCoerceable$|))
  (cond
   ((equal m mp) m)
   ((setq sl (|pmatch| mp m)) (sublis sl mp))
   ((|coerce| (list '|$fromCoerceable$| m env) mp) mp)
   (t nil))))
```

———-

6.6.8 defun coerceExit

[resolve p334]
[replaceExitEsc p??]
[coerce p325]
[$exitMode p??]

— defun coerceExit —

```
(defun |coerceExit| (arg1 mp)
 (let (x m e catchTag xp)
 (declare (special |$exitMode|))
  (setq x (first arg1))
  (setq m (second arg1))
  (setq e (third arg1))
  (setq mp (|resolve| m mp))
  (setq xp
    (|replaceExitEtc| x
      (setq catchTag (mkq (gensym))) '|TAGGEDexit| |$exitMode|))
  (|coerce| (list (list 'catch catchTag xp) m e) mp)))
```

———-

6.6.9 defplist compAtSign plist

We set up the `compAtSign` function to handle the `@` keyword by setting the `special` keyword on the `@` symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|@| 'special) 'compAtSign))
```

———-

6.6.10 defun compAtSign

[addDomain p233]
[comp p530]
[coerce p325]

— defun compAtSign —

```
(defun compAtSign (form mode env)
  (let ((newform (second form)) (mprime (third form)) tmp)
   (setq env (|addDomain| mprime env))
   (when (setq tmp (|comp| newform mprime env)) (|coerce| tmp mode))))
```

———-

6.6.11 defplist compCoerce plist

We set up the compCoerce function to handle the :: keyword by setting the special keyword on the :: symbol property list.

— postvars —

```
(eval-when (eval load)
 (setf (get '|::| 'special) '|compCoerce|))
```

———-

6.6.12 defun compCoerce

[addDomain p233]
[getmode p??]
[compCoerce1 p332]
[coerce p325]

— defun compCoerce —

```
(defun |compCoerce| (form mode env)
 (let (newform newmode tmp1 tmp4 z td)
  (setq newform (second form))
  (setq newmode (third form))
  (setq env (|addDomain| newmode env))
  (setq tmp1 (|getmode| newmode env))
  (cond
   ((setq td (|compCoerce1| newform newmode env))
    (|coerce| td mode))
   ((and (consp tmp1) (eq (qfirst tmp1) '|Mapping|)
         (consp (qrest tmp1)) (eq (qcddr tmp1) nil)
         (consp (qsecond tmp1))
         (eq (qcaadr tmp1) '|UnionCategory|))
    (setq z (qcdadr tmp1))
    (when
     (setq td
      (dolist (mode1 z tmp4)
```

```
      (setq tmp4 (or tmp4 (|compCoerce1| newform mode1 env)))))
   (|coerce| (list (car td) newmode (third td)) mode))))))
```

———-

6.6.13 defun compCoerce1

[comp p530]
[resolve p334]
[coerce p325]
[coerceByModemap p332]
[mkq p??]

— defun compCoerce1 —

```
(defun |compCoerce1| (form mode env)
 (let (m1 td tp gg pred code)
 (declare (special |$String| |$EmptyMode|))
  (when (setq td (or (|comp| form mode env) (|comp| form |$EmptyMode| env)))
    (setq m1 (if (stringp (second td)) |$String| (second td)))
    (setq mode (|resolve| m1 mode))
    (setq td (list (car td) m1 (third td)))
    (cond
     ((setq tp (|coerce| td mode)) tp)
     ((setq tp (|coerceByModemap| td mode)) tp)
     ((setq pred (|isSubset| mode (second td) env))
       (setq gg (gensym))
       (setq pred (subst gg '* pred :test #'equal))
       (setq code
        (list 'prog1
         (list 'let gg (first td))
          (cons '|check-subtype| (cons pred (list (mkq mode) gg)))))
       (list code mode (third td)))))))
```

———-

6.6.14 defun coerceByModemap

[modeEqual p335]
[isSubset p??]
[genDeltaEntry p??]

— defun coerceByModemap —

```
(defun |coerceByModemap| (arg1 mp)
 (let (x m env map cexpr u mm fn)
  (setq x (first arg1))
  (setq m (second arg1))
  (setq env (third arg1))
  (setq u
   (loop for modemap in (|getModemapList| '|coerce| 1 env)
```

```
    do
      (setq map (first modemap))
      (setq cexpr (second modemap))
    when
      (and (consp map) (consp (qrest map))
           (consp (qcddr map))
           (eq (qcdddr map) nil)
           (or (|modeEqual| (second map) mp) (|isSubset| (second map) mp env))
           (or (|modeEqual| (third map) m) (|isSubset| m (third map) env)))
    collect modemap))
  (when u
   (setq mm (first u))
   (setq fn (|genDeltaEntry| (cons '|coerce| mm)))
   (list (list '|call| fn x) mp env))))
```

———-

6.6.15 defun autoCoerceByModemap

[getModemapList p243]
[modeEqual p335]
[member p??]
[get p??]
[stackMessage p??]
[$fromCoerceable p??]

— defun autoCoerceByModemap —

```
(defun |autoCoerceByModemap| (arg1 target)
 (let (x source e map cexpr u fn y)
 (declare (special |$fromCoerceable$|))
  (setq x (first arg1))
  (setq source (second arg1))
  (setq e (third arg1))
  (setq u
   (loop for modemap in (|getModemapList| '|autoCoerce| 1 e)
    do
     (setq map (first modemap))
     (setq cexpr (second modemap))
    when
     (and (consp map) (consp (qrest map)) (consp (qcddr map))
                 (eq (qcdddr map) nil)
                 (|modeEqual| (second map) target)
                 (|modeEqual| (third map) source))
    collect cexpr))
  (when u
   (setq fn
    (let (result)
     (loop for item in u
      do
       (when (first item) (setq result (or result (second item)))))
     result))
```

```
    (when fn
    (cond
     ((and (consp source) (eq (qfirst source) '|Union|)
           (|member| target (qrest source)))
       (cond
        ((and (setq y (|get| x '|condition| e))
              (let (result)
               (loop for u in y do
                 (setq result
                   (or result
                    (and (consp u) (eq (qfirst u) '|case|) (consp (qrest u))
                         (consp (qcddr u))
                         (eq (qcdddr u) nil)
                         (equal (qthird u) target)))))
               result))
          (list (list '|call| fn x) target e))
        ((eq x '|$fromCoerceable$|) nil)
        (t
         (|stackMessage|
          (list '|cannot coerce: | x '|%l| '|      of mode: | source
                '|%l| '|      to: | target '| without a case statement|)))))
     (t
       (list (list '|call| fn x) target e)))))))
```

6.6.16 defun resolve

[modeEqual p335]
[mkUnion p335]
[$String p320]
[$EmptyMode p166]
[$NoValueMode p165]

— defun resolve —

```
(defun |resolve| (din dout)
 (declare (special |$String| |$EmptyMode| |$NoValueMode|))
 (cond
  ((or (equal din |$NoValueMode|) (equal dout |$NoValueMode|)) |$NoValueMode|)
  ((equal dout |$EmptyMode|) din)
  ((and (not (equal din dout)) (or (stringp din) (stringp dout)))
    (cond
       ((|modeEqual| dout |$String|) dout)
       ((|modeEqual| din |$String|) nil)
       (t (|mkUnion| din dout))))
    (t dout)))
```

6.6.17 defun mkUnion

[union p??]
[$Rep p??]

— defun mkUnion —

```
(defun |mkUnion| (a b)
 (declare (special |$Rep|))
  (cond
   ((and (eq b '$) (consp |$Rep|) (eq (qfirst |$Rep|) '|Union|))
     (qrest |$Rep|))
   ((and (consp a) (eq (qfirst a) '|Union|))
     (cond
      ((and (consp b) (eq (qfirst b) '|Union|))
        (cons '|Union| (|union| (qrest a) (qrest b))))
      (t (cons '|Union| (|union| (list b) (qrest a))))))
   ((and (consp b) (eq (qfirst b) '|Union|))
     (cons '|Union| (|union| (list a) (qrest b))))
   (t (list '|Union| a b))))
```

———-

6.6.18 defun This orders Unions

This orders Unions

— defun modeEqual —

```
(defun |modeEqual| (x y)
 (let (xl yl)
  (cond
   ((or (atom x) (atom y)) (equal x y))
   ((not (eql (|#| x) (|#| y))) nil)
   ((and (consp x) (eq (qfirst x) '|Union|) (consp y) (eq (qfirst y) '|Union|))
    (setq xl (qrest x))
    (setq yl (qrest y))
    (loop for a in xl do
     (loop for b in yl do
       (when (|modeEqual| a b)
         (setq xl (|delete| a xl))
         (setq yl (|delete| b yl))
         (return nil))))
    (unless (or xl yl) t))
  (t
   (let ((result t))
    (loop for u in x for v in y
     do (setq result (and result (|modeEqual| u v))))
    result)))))
```

———-

6.6.19 defun modeEqualSubst

[modeEqual p335]
[modeEqualSubst p336]
[length p??]

— defun modeEqualSubst —

```
(defun |modeEqualSubst| (m1 m env)
 (let (mp op z1 z2)
  (cond
   ((|modeEqual| m1 m) t)
   ((atom m1)
    (when (setq mp (car (|get| m1 '|value| env)))
     (|modeEqual| mp m)))
   ((and (consp m1) (consp m) (equal (qfirst m) (qfirst m1))
         (equal (|#| (qrest m1)) (|#| (qrest m))))
      (setq op (qfirst m1))
      (setq z1 (qrest m1))
      (setq z2 (qrest m))
      (let ((result t))
       (loop for xm1 in z1 for xm2 in z2
        do (setq result (and result (|modeEqualSubst| xm1 xm2 env))))
       result))
   (t nil))))
```

———

Chapter 7

Post Transformers

7.1 Direct called postparse routines

7.1.1 defun postTransform

[postTran p338]
[postTransform identp (vol5)]
[postTransformCheck p340]
[aplTran p368]

— defun postTransform —

```
(defun postTransform (y)
 (let (x tmp1 tmp2 tmp3 tmp4 tmp5 tt l u)
  (setq x y)
  (setq u (|postTran| x))
  (when
   (and (consp u) (eq (qfirst u) '|@Tuple|)
        (progn
         (setq tmp1 (qrest u))
         (and (consp tmp1)
              (progn (setq tmp2 (reverse tmp1)) t)
              (consp tmp2)
              (progn
               (setq tmp3 (qfirst tmp2))
               (and (consp tmp3)
                    (eq (qfirst tmp3) '|:|)
                    (progn
                     (setq tmp4 (qrest tmp3))
                     (and (consp tmp4)
                          (progn
                           (setq y (qfirst tmp4))
                           (setq tmp5 (qrest tmp4))
                           (and (consp tmp5)
                                (eq (qrest tmp5) nil)
                                (progn (setq tt (qfirst tmp5)) t)))))))
                        (progn (setq l (qrest tmp2)) t)
```

```
                        (progn (setq l (nreverse l)) t)))
              (dolist (x l t) (unless (identp x) (return nil))))
    (setq u (list '|:| (cons 'listof (append l (list y))) tt)))
  (postTransformCheck u)
  (aplTran u)))
```

———-

7.1.2 defun postTran

[postAtom p339]
[postTran p338]
[unTuple p375]
[postTranList p339]
[postForm p341]
[postOp p339]
[postScriptsForm p339]

— **defun postTran** —

```
(defun |postTran| (x)
 (let (op f tmp1 a tmp2 tmp3 b y)
  (if (atom x)
   (postAtom x)
   (progn
    (setq op (car x))
    (cond
     ((and (atom op) (setq f (getl op '|postTran|)))
      (funcall f x))
     ((and (consp op) (eq (qfirst op) '|elt|)
           (progn
            (setq tmp1 (qrest op))
            (and (consp tmp1)
                 (progn
                   (setq a (qfirst tmp1))
                   (setq tmp2 (qrest tmp1))
                   (and (consp tmp2)
                        (eq (qrest tmp2) nil)
                        (progn (setq b (qfirst tmp2)) t))))))
      (cons (|postTran| op) (cdr (|postTran| (cons b (cdr x))))))
     ((and (consp op) (eq (qfirst op) '|Scripts|))
      (postScriptsForm op
       (dolist (y (rest x) tmp3)
        (setq tmp3 (append tmp3 (|unTuple| (|postTran| y)))))))
     ((not (equal op (setq y (postOp op))))
      (cons y (postTranList (cdr x))))
     (t (postForm x)))))))
```

———-

7.1.3 defun postOp

— defun postOp —

```
(defun postOp (x)
 (declare (special $boot))
 (cond
  ((eq x '|:=|) 'let)
  ((eq x '|:-|) 'letd)
  ((eq x '|Attribute|) 'attribute)
  (t x)))
```

———-

7.1.4 defun postAtom

[$boot p??]

— defun postAtom —

```
(defun postAtom (x)
 (declare (special $boot))
  (cond
    ($boot x)
    ((eql x 0) '(|Zero|))
    ((eql x 1) '(|One|))
    ((eq x t) 't$)
    ((and (identp x) (getdatabase x 'niladic)) (list x))
    (t x)))
```

———-

7.1.5 defun postTranList

[postTran p338]

— defun postTranList —

```
(defun postTranList (x)
 (loop for y in x collect (|postTran| y)))
```

———-

7.1.6 defun postScriptsForm

[getScriptName p371]
[length p??]
[postTranScripts p340]

— defun postScriptsForm —

```
(defun postScriptsForm (form argl)
 (let ((op (second form)) (a (third form)))
  (cons (getScriptName op a (|#| argl))
        (append (postTranScripts a) argl))))
```

———-

7.1.7 defun postTranScripts

[postTranScripts p340]
[postTran p338]

— defun postTranScripts —

```
(defun postTranScripts (a)
 (labels (
  (fn (x)
   (if (and (consp x) (eq (qfirst x) '|@Tuple|))
    (qrest x)
    (list x))))
 (let (tmp1 tmp2 tmp3)
  (cond
   ((and (consp a) (eq (qfirst a) '|PrefixSC|)
         (progn
          (setq tmp1 (qrest a))
          (and (consp tmp1) (eq (qrest tmp1) nil))))
     (postTranScripts (qfirst tmp1)))
   ((and (consp a) (eq (qfirst a) '|;|))
    (dolist (y (qrest a) tmp2)
     (setq tmp2 (append tmp2 (postTranScripts y)))))
   ((and (consp a) (eq (qfirst a) '|,|))
    (dolist (y (qrest a) tmp3)
     (setq tmp3 (append tmp3 (fn (|postTran| y))))))
   (t (list (|postTran| a)))))))
```

———-

7.1.8 defun postTransformCheck

[postcheck p341]
[$defOp p??]

— defun postTransformCheck —

```
(defun postTransformCheck (x)
 (let (|$defOp|)
 (declare (special |$defOp|))
  (setq |$defOp| nil)
  (postcheck x)))
```

———-

7.1.9 defun postcheck

[setDefOp p368]
[postcheck p341]

— defun postcheck —

```
(defun postcheck (x)
 (cond
  ((atom x) nil)
  ((and (consp x) (eq (qfirst x) 'def) (consp (qrest x)))
   (setDefOp  (qsecond x))
   (postcheck (qcddr x)))
  ((and (consp x) (eq (qfirst x) 'quote)) nil)
  (t (postcheck (car x)) (postcheck (cdr x)))))
```

———

7.1.10 defun postError

[bumperrorcount p490]
[$defOp p??]
[$InteractiveMode p??]
[$postStack p??]

— defun postError —

```
(defun postError (msg)
 (let (xmsg)
  (declare (special |$defOp| |$postStack| |$InteractiveMode|))
  (bumperrorcount '|precompilation|)
  (setq xmsg
   (if (and (not (eq |$defOp| '|$defOp|)) (null |$InteractiveMode|))
    (cons |$defOp| (cons ": " msg))
    msg))
  (push xmsg |$postStack|)
  nil))
```

———

7.1.11 defun postForm

[postTranList p339]
[internl p??]
[postTran p338]
[postError p341]
[bright p??]
[$boot p??]

— defun postForm —

```
(defun postForm (u)
 (let (op argl arglp numOfArgs opp x)
 (declare (special $boot))
  (seq
   (setq op (car u))
   (setq argl (cdr u))
   (setq x
    (cond
     ((atom op)
       (setq arglp (postTranList argl))
       (setq opp
         (seq
          (exit op)
          (when $boot (exit op))
          (when (or (getl op '|Led|)  (getl op '|Nud|) (eq op 'in)) (exit op))
          (setq numOfArgs
           (cond
            ((and (consp arglp) (eq (qrest arglp) nil) (consp (qfirst arglp))
                  (eq (qcaar arglp) '|@Tuple|))
              (|#| (qcdar arglp)))
            (t 1)))
          (internl '* (princ-to-string numOfArgs) (pname op))))
       (cons opp arglp))
     ((and (consp op) (eq (qfirst op) '|Scripts|))
       (append (|postTran| op) (postTranList argl)))
     (t
      (setq u (postTranList u))
      (cond
       ((and (consp u) (consp (qfirst u)) (eq (qcaar u) '|@Tuple|))
         (postError
          (cons "  "
           (append (|bright| u)
            (list "is illegal because tuples cannot be applied!" '|%l|
                  "   Did you misuse infix dot?"))))))
       u)))
   (cond
    ((and (consp x) (consp (qrest x)) (eq (qcddr x) nil)
          (consp (qsecond x)) (eq (qcaadr x) '|@Tuple|))
     (cons (car x) (qcdadr x)))
    (t x)))))
```

———

7.2 Indirect called postparse routines

In the **postTran** function there is the code:

```
((and (atom op) (setq f (getl op '|postTran|)))
 (funcall f x))
```

The functions in this section are called through the symbol-plist of the symbol being parsed. The original list read:

```
add            postAdd
@              postAtSign
:BF:           postBigFloat
Block          postBlock
CATEGORY       postCategory
COLLECT        postCollect
:              postColon
::             postColonColon
,              postComma
construct      postConstruct
==             postDef
=>             postExit
if             postIf
in             postin       ;" the infix operator version of in"
IN             postIn       ;" the iterator form of in"
Join           postJoin
->             postMapping
==>            postMDef
pretend        postPretend
QUOTE          postQUOTE
Reduce         postReduce
REPEAT         postRepeat
Scripts        postScripts
;              postSemiColon
Signature      postSignature
/              postSlash
@Tuple         postTuple
TupleCollect   postTupleCollect
where          postWhere
with           postWith
```

7.2.1 defplist postAdd plist

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|add| '|postTran|) '|postAdd|))
```

————-

7.2.2 defun postAdd

[postTran p338]
[postCapsule p344]

— **defun postAdd** —

```
(defun |postAdd| (arg)
 (if (null (cddr arg))
  (|postCapsule| (second arg))
  (list '|add| (|postTran| (second arg)) (|postCapsule| (third arg)))))
```

———-

7.2.3 defun postCapsule

[checkWarning p494]
[postBlockItem p344]
[postBlockItemList p344]
[postFlatten p352]

— defun postCapsule —

```
(defun |postCapsule| (x)
 (let (op)
  (cond
   ((null (and (consp x) (progn (setq op (qfirst x)) t)))
     (|checkWarning| (list "Apparent indentation error following add")))
   ((or (integerp op) (eq op '==))
     (list 'capsule (|postBlockItem| x)))
   ((eq op '|;|)
     (cons 'capsule (|postBlockItemList| (|postFlatten| x '|;|))))
   ((eq op '|if|)
     (list 'capsule (|postBlockItem| x)))
   (t (|checkWarning| (list "Apparent indentation error following add"))))))
```

———-

7.2.4 defun postBlockItemList

[postBlockItem p344]

— defun postBlockItemList —

```
(defun |postBlockItemList| (args)
 (let (result)
  (dolist (item args (nreverse result))
   (push (|postBlockItem| item) result))))
```

———-

7.2.5 defun postBlockItem

[postTran p338]

— defun postBlockItem —

```
(defun |postBlockItem| (x)
 (let ((tmp1 t) tmp2 y tt z)
  (setq x (|postTran| x))
  (if
   (and (consp x) (eq (qfirst x) '|@Tuple|)
```

```
     (progn
       (and (consp (qrest x))
            (progn (setq tmp2 (reverse (qrest x))) t)
            (consp tmp2)
            (progn
             (and (consp (qfirst tmp2)) (eq (qcaar tmp2) '|:|)
             (progn
              (and (consp (qcdar tmp2))
                   (progn
                    (setq y (qcadar tmp2))
                    (and (consp (qcddar tmp2))
                         (eq (qcdddar tmp2) nil)
                    (progn (setq tt (qcaddar tmp2)) t)))))))
            (progn (setq z (qrest tmp2)) t)
            (progn (setq z (nreverse z)) T)))
      (do ((tmp6 nil (null tmp1)) (tmp7 z (cdr tmp7)) (x nil))
          ((or tmp6 (atom tmp7)) tmp1)
        (setq x (car tmp7))
        (setq tmp1 (and tmp1 (identp x)))))
     (cons '|:| (cons (cons 'listof (append z (list y))) (list tt)))
     x)))
```

———-

7.2.6 defplist postAtSign plist

— postvars —

```
(eval-when (eval load)
 (setf (get '@ '|postTran|) '|postAtSign|))
```

———-

7.2.7 defun postAtSign

[postTran p338]
[postType p345]

— defun postAtSign —

```
(defun |postAtSign| (arg)
 (cons '@ (cons (|postTran| (second arg)) (|postType| (third arg)))))
```

———-

7.2.8 defun postType

[postTran p338]
[unTuple p375]

— defun postType —

```
(defun |postType| (typ)
 (let (source target)
  (cond
   ((and (consp typ) (eq (qfirst typ) '->) (consp (qrest typ))
         (consp (qcddr typ)) (eq (qcdddr typ) nil))
     (setq source (qsecond typ))
     (setq target (qthird typ))
     (cond
      ((eq source '|constant|)
        (list (list (|postTran| target)) '|constant|))
      (t
        (list (cons '|Mapping|
                (cons (|postTran| target)
                      (|unTuple| (|postTran| source))))))))
   ((and (consp typ) (eq (qfirst typ) '->)
         (consp (qrest typ)) (eq (qcddr typ) nil))
      (list (list '|Mapping| (|postTran| (qsecond typ)))))
   (t (list (|postTran| typ))))))
```

———-

7.2.9 defplist postBigFloat plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|:BF:| '|postTran|) '|postBigFloat|))
```

———-

7.2.10 defun postBigFloat

[postTran p338]
[$boot p??]
[$InteractiveMode p??]

— defun postBigFloat —

```
(defun |postBigFloat| (arg)
 (let (mant expon eltword)
  (declare (special $boot |$InteractiveMode|))
  (setq mant (second arg))
  (setq expon (cddr arg))
  (if $boot
   (times (float mant) (expt (float 10) expon))
   (progn
    (setq eltword (if |$InteractiveMode| '|$elt| '|elt|))
    (|postTran|
```

```
    (list (list eltword '(|Float|) '|float|)
      (list '|,| (list '|,| mant expon) 10)))))))
```

———-

7.2.11 defplist postBlock plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|Block| '|postTran|) '|postBlock|))
```

———-

7.2.12 defun postBlock

[postBlockItemList p344]
[postTran p338]

— defun postBlock —

```
(defun |postBlock| (arg)
 (let (tmp1 x y)
  (setq tmp1 (reverse (cdr arg)))
  (setq x (car tmp1))
  (setq y (nreverse (cdr tmp1)))
  (cons 'seq
   (append (|postBlockItemList| y) (list (list '|exit| (|postTran| x)))))))
```

———-

7.2.13 defplist postCategory plist

— postvars —

```
(eval-when (eval load)
 (setf (get 'category '|postTran|) '|postCategory|))
```

———-

7.2.14 defun postCategory

[postTran p338]
[nreverse0 p??]
[$insidePostCategoryIfTrue p??]

— defun postCategory —

```
(defun |postCategory| (u)
 (declare (special |$insidePostCategoryIfTrue|))
 (labels (
  (fn (arg)
   (let (|$insidePostCategoryIfTrue|)
   (declare (special |$insidePostCategoryIfTrue|))
    (setq |$insidePostCategoryIfTrue| t)
    (|postTran| arg))) )
 (let ((z (cdr u)) op tmp1)
  (if (null z)
   u
   (progn
     (setq op (if |$insidePostCategoryIfTrue| 'progn 'category))
     (cons op (dolist (x z (nreverse0 tmp1)) (push (fn x) tmp1))))))))
```

———

7.2.15 defun postCollect,finish

[postMakeCons p349]
[tuple2List p494]
[postTranList p339]

— defun postCollect,finish —

```
(defun |postCollect,finish| (op itl y)
 (let (tmp2 tmp5 newBody)
  (cond
   ((and (consp y) (eq (qfirst y) '|:|)
         (consp (qrest y)) (eq (qcddr y) nil))
    (list 'reduce '|append| 0 (cons op (append itl (list (qsecond y))))))
   ((and (consp y) (eq (qfirst y) '|Tuple|))
    (setq newBody
     (cond
      ((dolist (x (qrest y) tmp2)
        (setq tmp2
         (or tmp2 (and (consp x) (eq (qfirst x) '|:|)
                       (consp (qrest x)) (eq (qcddr x) nil)))))
       (|postMakeCons| (qrest y)))
      ((dolist (x (qrest y) tmp5)
         (setq tmp5 (or tmp5 (and (consp x) (eq (qfirst x) 'segment)))))
        (|tuple2List| (qrest y)))
      (t (cons '|construct| (postTranList (qrest y))))))
     (list 'reduce '|append| 0 (cons op (append itl (list newBody)))))
   (t (cons op (append itl (list y)))))))
```

———

7.2.16 defun postMakeCons

[postMakeCons p349]
[postTran p338]

— defun postMakeCons —

```
(defun |postMakeCons| (args)
 (let (a b)
  (cond
   ((null args) '|nil|)
   ((and (consp args) (consp (qfirst args)) (eq (qcaar args) '|:|)
         (consp (qcdar args)) (eq (qcddar args) nil))
     (setq a (qcadar args))
     (setq b (qrest args))
     (if b
       (list '|append| (|postTran| a) (|postMakeCons| b))
       (|postTran| a)))
   (t (list '|cons| (|postTran| (car args)) (|postMakeCons| (cdr args)))))))
```

———-

7.2.17 defplist postCollect plist

— postvars —

```
(eval-when (eval load)
 (setf (get 'collect '|postTran|) '|postCollect|))
```

———-

7.2.18 defun postCollect

[postCollect,finish p348]
[postCollect p349]
[postIteratorList p350]
[postTran p338]

— defun postCollect —

```
(defun |postCollect| (arg)
 (let (constructOp tmp3 m itl x)
  (setq constructOp (car arg))
  (setq tmp3 (reverse (cdr arg)))
  (setq x (car tmp3))
  (setq m (nreverse (cdr tmp3)))
  (cond
   ((and (consp x) (consp (qfirst x)) (eq (qcaar x) '|elt|)
         (consp (qcdar x)) (consp (qcddar x))
         (eq (qcdddar x) nil)
         (eq (qcaddar x) '|construct|))
```

```
    (|postCollect|
     (cons (list '|elt| (qcadar x) 'collect)
      (append m (list (cons '|construct| (qrest x)))))))
   (t
    (setq itl (|postIteratorList| m))
    (setq x
     (if (and (consp x) (eq (qfirst x) '|construct|)
              (consp (qrest x)) (eq (qcddr x) nil))
       (qsecond x)
       x))
    (|postCollect,finish| constructOp itl (|postTran| x))))))
```

———-

7.2.19 defun postIteratorList

[postTran p338]
[postInSeq p357]
[postIteratorList p350]

— defun postIteratorList —

```
(defun |postIteratorList| (args)
 (let (z p y u a b)
  (cond
   ((consp args)
     (setq p (|postTran| (qfirst args)))
     (setq z (qrest args))
     (cond
       ((and (consp p) (eq (qfirst p) 'in) (consp (qrest p))
             (consp (qcddr p)) (eq (qcdddr p) nil))
        (setq y (qsecond p))
        (setq u (qthird p))
        (cond
          ((and (consp u) (eq (qfirst u) '|\||) (consp (qrest u))
                (consp (qcddr u)) (eq (qcdddr u) nil))
            (setq a (qsecond u))
            (setq b (qthird u))
            (cons (list 'in y (|postInSeq| a))
              (cons (list '|\|| b)
                (|postIteratorList| z))))
          (t (cons (list 'in y (|postInSeq| u)) (|postIteratorList| z)))))
       (t (cons p (|postIteratorList| z)))))
   (t args))))
```

———-

7.2.20 defplist postColon plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|:| '|postTran|) '|postColon|))
```

———-

7.2.21 defun postColon

[postTran p338]
[postType p345]

— defun postColon —

```
(defun |postColon| (u)
 (cond
  ((and (consp u) (eq (qfirst u) '|:|)
        (consp (qrest u)) (eq (qcddr u) nil))
    (list '|:| (|postTran| (qsecond u))))
  ((and (consp u) (eq (qfirst u) '|:|) (consp (qrest u))
        (consp (qcddr u)) (eq (qcdddr u) nil))
   (cons '|:| (cons (|postTran| (second u)) (|postType| (third u)))))))
```

———-

7.2.22 defplist postColonColon plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|::| '|postTran|) '|postColonColon|))
```

———-

7.2.23 defun postColonColon

[postForm p341]
[$boot p??]

— defun postColonColon —

```
(defun |postColonColon| (u)
  (if (and $boot (consp u) (eq (qfirst u) '|::|) (consp (qrest u))
           (consp (qcddr u)) (eq (qcdddr u) nil))
   (intern (princ-to-string (third u)) (second u))
   (postForm u)))
```

———-

7.2.24 defplist postComma plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|,| '|postTran|) '|postComma|))
```

———

7.2.25 defun postComma

[postTuple p366]
[comma2Tuple p352]

— defun postComma —

```
(defun |postComma| (u)
 (|postTuple| (|comma2Tuple| u)))
```

———

7.2.26 defun comma2Tuple

[postFlatten p352]

— defun comma2Tuple —

```
(defun |comma2Tuple| (u)
 (cons '|@Tuple| (|postFlatten| u '|,|)))
```

———

7.2.27 defun postFlatten

[postFlatten p352]

— defun postFlatten —

```
(defun |postFlatten| (x op)
 (let (a b)
  (cond
   ((and (consp x) (equal (qfirst x) op) (consp (qrest x))
         (consp (qcddr x)) (eq (qcdddr x) nil))
     (setq a (qsecond x))
     (setq b (qthird x))
     (append (|postFlatten| a op) (|postFlatten| b op)))
   (t (list x)))))
```

———

7.2.28 defplist postConstruct plist

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|construct| '|postTran|) '|postConstruct|))
```

———-

7.2.29 defun postConstruct

[comma2Tuple p352]
[postTranSegment p354]
[postMakeCons p349]
[tuple2List p494]
[postTranList p339]
[postTran p338]

— **defun postConstruct** —

```
(defun |postConstruct| (u)
 (let (b a tmp4 tmp7)
  (cond
   ((and (consp u) (eq (qfirst u) '|construct|)
         (consp (qrest u)) (eq (qcddr u) nil))
     (setq b (qsecond u))
     (setq a
      (if (and (consp b) (eq (qfirst b) '|,|))
       (|comma2Tuple| b)
       b))
     (cond
      ((and (consp a) (eq (qfirst a) 'segment) (consp (qrest a))
            (consp (qcddr a)) (eq (qcdddr a) nil))
       (list '|construct| (|postTranSegment| (second a) (third a))))
      ((and (consp a) (eq (qfirst a) '|@Tuple|))
        (cond
         ((dolist (x (qrest a) tmp4)
           (setq tmp4
            (or tmp4
             (and (consp x) (eq (qfirst x) '|:|)
                  (consp (qrest x)) (eq (qcddr x) nil)))))
          (|postMakeCons| (qrest a)))
         ((dolist (x (qrest a) tmp7)
            (setq tmp7 (or tmp7 (and (consp x) (eq (qfirst x) 'segment)))))
          (|tuple2List| (qrest a)))
         (t (cons '|construct| (postTranList (qrest a))))))
      (t (list '|construct| (|postTran| a)))))
   (t u))))
```

———-

7.2.30 defun postTranSegment

[postTran p338]

— defun postTranSegment —

```
(defun |postTranSegment| (p q)
  (list 'segment (|postTran| p) (when q (|postTran| q))))
```

——–

7.2.31 defplist postDef plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|==| '|postTran|) '|postDef|))
```

——–

7.2.32 defun postDef

[postMDef p360]
[recordHeaderDocumentation p425]
[postTran p338]
[postDefArgs p355]
[nreverse0 p??]
[$boot p??]
[$maxSignatureLineNumber p??]
[$headerDocumentation p??]
[$docList p??]
[$InteractiveMode p??]

— defun postDef —

```
(defun |postDef| (arg)
 (let (defOp rhs lhs targetType tmp1 op argl newLhs
      argTypeList typeList form specialCaseForm tmp4 tmp6 tmp8)
 (declare (special $boot |$maxSignatureLineNumber| |$headerDocumentation|
                   |$docList| |$InteractiveMode|))
  (setq defOp (first arg))
  (setq lhs (second arg))
  (setq rhs (third arg))
  (if (and (consp lhs) (eq (qfirst lhs) '|macro|)
          (consp (qrest lhs)) (eq (qcddr lhs) nil))
   (|postMDef| (list '==> (second lhs) rhs))
   (progn
    (unless $boot (|recordHeaderDocumentation| nil))
    (when (not (eql |$maxSignatureLineNumber| 0))
      (setq |$docList|
       (cons (cons '|constructor| |$headerDocumentation|) |$docList|))
```

```
      (setq |$maxSignatureLineNumber| 0))
    (setq lhs (|postTran| lhs))
    (setq tmp1
     (if (and (consp lhs) (eq (qfirst lhs) '|:|)) (cdr lhs) (list lhs nil)))
    (setq form (first tmp1))
    (setq targetType (second tmp1))
    (when (and (null |$InteractiveMode|) (atom form)) (setq form (list form)))
    (setq newLhs
     (if (atom form)
      form
      (progn
       (setq tmp1
        (dolist (x form (nreverse0 tmp4))
         (push
           (if (and (consp x) (eq (qfirst x) '|:|) (consp (qrest x))
                    (consp (qcddr x)) (eq (qcdddr x) nil))
            (second x)
            x)
           tmp4)))
       (setq op (car tmp1))
       (setq argl (cdr tmp1))
       (cons op (|postDefArgs| argl)))))
    (setq argTypeList
     (unless (atom form)
      (dolist (x (cdr form) (nreverse0 tmp6))
       (push
         (when (and (consp x) (eq (qfirst x) '|:|) (consp (qrest x))
              (consp (qcddr x)) (eq (qcdddr x) nil))
           (third x))
         tmp6))))
    (setq typeList (cons targetType argTypeList))
    (when (atom form) (setq form (list form)))
    (setq specialCaseForm (dolist (x form (nreverse tmp8)) (push nil tmp8)))
    (list 'def newLhs typeList specialCaseForm (|postTran| rhs))))))
```

———-

7.2.33 defun postDefArgs

[postError p341]
[postDefArgs p355]

— defun postDefArgs —

```
(defun |postDefArgs| (args)
 (let (a b)
  (cond
   ((null args) args)
   ((and (consp args) (consp (qfirst args)) (eq (qcaar args) '|:|)
         (consp (qcdar args)) (eq (qcddar args) nil))
     (setq a (qcadar args))
     (setq b (qrest args))
     (cond
```

```
      (b (postError
          (list  "   Argument" a "of indefinite length must be last")))
      ((or (atom a) (and (consp a) (eq (qfirst a) 'quote)))
        a)
      (t
        (postError
          (list "   Argument" a "of indefinite length must be a name")))))
   (t (cons (car args) (|postDefArgs| (cdr args)))))))
```

———-

7.2.34 defplist postExit plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|=>| '|postTran|) '|postExit|))
```

———-

7.2.35 defun postExit

[postTran p338]

— defun postExit —

```
(defun |postExit| (arg)
 (list 'if (|postTran| (second arg))
  (list '|exit| (|postTran| (third arg)))
   '|noBranch|))
```

———-

7.2.36 defplist postIf plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|if| '|postTran|) '|postIf|))
```

———-

7.2.37 defun postIf

[nreverse0 p??]
[postTran p338]
[$boot p??]

— defun postIf —

```
(defun |postIf| (arg)
 (let (tmp1)
  (if (null (and (consp arg) (eq (qfirst arg) '|if|)))
   arg
   (cons 'if
    (dolist (x (qrest arg) (nreverse0 tmp1))
     (push
       (if (and (null (setq x (|postTran| x))) (null $boot)) '|noBranch| x)
       tmp1))))))
```

———-

7.2.38 defplist postin plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|in| '|postTran|) '|postin|))
```

———-

7.2.39 defun postin

[systemErrorHere p??]
[postTran p338]
[postInSeq p357]

— defun postin —

```
(defun |postin| (arg)
 (if (null (and (consp arg) (eq (qfirst arg) '|in|) (consp (qrest arg))
                (consp (qcddr arg)) (eq (qcdddr arg) nil)))
   (|systemErrorHere| "postin")
   (list '|in| (|postTran| (second arg)) (|postInSeq| (third arg)))))
```

———-

7.2.40 defun postInSeq

[postTranSegment p354]
[tuple2List p494]
[postTran p338]

— defun postInSeq —

```
(defun |postInSeq| (seq)
 (cond
```

```
  ((and (consp seq) (eq (qfirst seq) 'segment) (consp (qrest seq))
        (consp (qcddr seq)) (eq (qcdddr seq) nil))
     (|postTranSegment| (second seq) (third seq)))
  ((and (consp seq) (eq (qfirst seq) '|@Tuple|))
     (|tuple2List| (qrest seq)))
  (t (|postTran| seq))))
```

———-

7.2.41 defplist postIn plist

— postvars —

```
(eval-when (eval load)
 (setf (get 'in '|postTran|) '|postIn|))
```

———-

7.2.42 defun postIn

[systemErrorHere p??]
[postTran p338]
[postInSeq p357]

— defun postIn —

```
(defun |postIn| (arg)
  (if (null (and (consp arg) (eq (qfirst arg) 'in) (consp (qrest arg))
                 (consp (qcddr arg)) (eq (qcdddr arg) nil)))
    (|systemErrorHere| "postIn")
    (list 'in (|postTran| (second arg)) (|postInSeq| (third arg)))))
```

———-

7.2.43 defplist postJoin plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|Join| '|postTran|) '|postJoin|))
```

———-

7.2.44 defun postJoin

[postTran p338]
[postTranList p339]

— **defun postJoin** —

```
(defun |postJoin| (arg)
 (let (a l al)
  (setq a (|postTran| (cadr arg)))
  (setq l (postTranList (cddr arg)))
  (when (and (consp l) (eq (qrest l) nil) (consp (qfirst l))
             (member (qcaar l) '(attribute signature)))
    (setq l (list (list 'category (qfirst l)))))
  (setq al (if (and (consp a) (eq (qfirst a) '|@Tuple|)) (qrest a) (list a)))
  (cons '|Join| (append al l))))
```

———-

7.2.45 defplist postMapping plist

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|->| '|postTran|) '|postMapping|))
```

———-

7.2.46 defun postMapping

[postTran p338]
[unTuple p375]

— **defun postMapping** —

```
(defun |postMapping| (u)
 (if (null (and (consp u) (eq (qfirst u) '->) (consp (qrest u))
           (consp (qcddr u)) (eq (qcdddr u) nil)))
   u
   (cons '|Mapping|
    (cons (|postTran| (third u))
          (|unTuple| (|postTran| (second u)))))))
```

———-

7.2.47 defplist postMDef plist

— **postvars** —

```
(eval-when (eval load)
 (setf (get '|==>| '|postTran|) '|postMDef|))
```

———-

7.2.48 defun postMDef

[postTran p338]
[throwkeyedmsg p??]
[nreverse0 p??]
[$InteractiveMode p??]
[$boot p??]

— defun postMDef —

```
(defun |postMDef| (arg)
 (let (rhs lhs tmp1 targetType form newLhs typeList tmp4 tmp5 tmp8)
 (declare (special |$InteractiveMode| $boot))
  (setq lhs (second arg))
  (setq rhs (third arg))
  (cond
   ((and |$InteractiveMode| (null $boot))
    (setq lhs (|postTran| lhs))
    (if (null (identp lhs))
      (|throwkeyedmsg| "The left-hand side of a => form must be a symbol." nil)
      (list 'mdef lhs nil nil (|postTran| rhs))))
   (t
    (setq lhs (|postTran| lhs))
    (setq tmp1
     (if (and (consp lhs) (eq (qfirst lhs) '|:|)) (cdr lhs) (list lhs nil)))
    (setq form (first tmp1))
    (setq targetType (second tmp1))
    (setq form (if (atom form) (list form) form))
    (setq newLhs
     (dolist (x form (nreverse0 tmp4))
      (push
       (if (and (consp x) (eq (qfirst x) '|:|) (consp (qrest x))) (second x) x)
       tmp4)))
    (setq typeList
     (cons targetType
      (dolist (x (qrest form) (nreverse0 tmp5))
       (push
         (when (and (consp x) (eq (qfirst x) '|:|) (consp (qrest x))
                    (consp (qcddr x)) (eq (qcdddr x) nil))
            (third x))
          tmp5))))
    (list 'mdef newLhs typeList
     (dolist (x form (nreverse0 tmp8)) (push nil tmp8))
     (|postTran| rhs))))))
```

———-

7.2.49 defplist postPretend plist

— postvars —

```
(eval-when (eval load)
```

```
 (setf (get '|pretend| '|postTran|) '|postPretend|))
```

———-

7.2.50 defun postPretend

[postTran p338]
[postType p345]

— defun postPretend —

```
(defun |postPretend| (arg)
 (cons '|pretend| (cons (|postTran| (second arg)) (|postType| (third arg)))))
```

———-

7.2.51 defplist postQUOTE plist

— postvars —

```
(eval-when (eval load)
 (setf (get 'quote '|postTran|) '|postQUOTE|))
```

———-

7.2.52 defun postQUOTE

— defun postQUOTE —

```
(defun |postQUOTE| (arg) arg)
```

———-

7.2.53 defplist postReduce plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|Reduce| '|postTran|) '|postReduce|))
```

———-

7.2.54 defun postReduce

[postTran p338]
[postReduce p361]

[$InteractiveMode p??]

— defun postReduce —

```
(defun |postReduce| (arg)
 (let (op expr g)
  (setq op (second arg))
  (setq expr (third arg))
  (if (or |$InteractiveMode| (and (consp expr) (eq (qfirst expr) 'collect)))
    (list 'reduce op 0 (|postTran| expr))
    (|postReduce|
     (list '|Reduce| op
      (list 'collect
       (list 'in (setq g (gensym)) expr)
        (list '|construct| g)))))))
```

———

7.2.55 defplist postRepeat plist

— postvars —

```
(eval-when (eval load)
 (setf (get 'repeat '|postTran|) '|postRepeat|))
```

———

7.2.56 defun postRepeat

[postIteratorList p350]
[postTran p338]

— defun postRepeat —

```
(defun |postRepeat| (arg)
 (let (tmp1 x m)
  (setq tmp1 (reverse (cdr arg)))
  (setq x (car tmp1))
  (setq m (nreverse (cdr tmp1)))
  (cons 'repeat (append (|postIteratorList| m) (list (|postTran| x))))))
```

———

7.2.57 defplist postScripts plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|Scripts| '|postTran|) '|postScripts|))
```

———-

7.2.58 defun postScripts

[getScriptName p371]
[postTranScripts p340]

— defun postScripts —

```
(defun |postScripts| (arg)
 (cons (getScriptName (second arg) (third arg) 0)
       (postTranScripts (third arg))))
```

———-

7.2.59 defplist postSemiColon plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|;| '|postTran|) '|postSemiColon|))
```

———-

7.2.60 defun postSemiColon

[postBlock p347]
[postFlattenLeft p363]

— defun postSemiColon —

```
(defun |postSemiColon| (u)
 (|postBlock| (cons '|Block| (|postFlattenLeft| u '|;|))))
```

———-

7.2.61 defun postFlattenLeft

[postFlattenLeft p363]

— defun postFlattenLeft —

```
(defun |postFlattenLeft| (x op)
 (let (a b)
  (cond
   ((and (consp x) (equal (qfirst x) op) (consp (qrest x))
         (consp (qcddr x)) (eq (qcdddr x) nil))
     (setq a (qsecond x))
```

```
    (setq b (qthird x))
    (append (|postFlattenLeft| a op) (list b)))
  (t (list x)))))
```

———-

7.2.62 defplist postSignature plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|Signature| '|postTran|) '|postSignature|))
```

———-

7.2.63 defun postSignature

[postType p345]
[removeSuperfluousMapping p364]
[killColons p365]

— defun postSignature —

```
(defun |postSignature| (arg)
 (let (sig sig1 op)
  (setq op (second arg))
  (setq sig (third arg))
  (when (and (consp sig) (eq (qfirst sig) '->))
    (setq sig1 (|postType| sig))
    (setq op (postAtom (if (stringp op) (setq op (intern op)) op)))
    (cons 'signature
     (cons op (|removeSuperfluousMapping| (|killColons| sig1)))))))
```

———-

7.2.64 defun removeSuperfluousMapping

— defun removeSuperfluousMapping —

```
(defun |removeSuperfluousMapping| (sig1)
 (if (and (consp sig1) (consp (qfirst sig1)) (eq (qcaar sig1) '|Mapping|))
  (cons (cdr (qfirst sig1)) (qrest sig1))
  sig1))
```

———-

7.2.65 defun killColons

[killColons p365]

— defun killColons —

```
(defun |killColons| (x)
 (cond
  ((atom x) x)
  ((and (consp x) (eq (qfirst x) '|Record|)) x)
  ((and (consp x) (eq (qfirst x) '|Union|)) x)
  ((and (consp x) (eq (qfirst x) '|:|) (consp (qrest x))
        (consp (qcddr x)) (eq (qcdddr x) nil))
   (|killColons| (third x)))
  (t (cons (|killColons| (car x)) (|killColons| (cdr x))))))
```

———-

7.2.66 defplist postSlash plist

— postvars —

```
(eval-when (eval load)
 (setf (get '/ '|postTran|) '|postSlash|))
```

———-

7.2.67 defun postSlash

[postTran p338]

— defun postSlash —

```
(defun |postSlash| (arg)
  (if (stringp (second arg))
   (|postTran| (list '|Reduce| (intern (second arg)) (third arg) ))
   (list '/ (|postTran| (second arg)) (|postTran| (third arg)))))
```

———-

7.2.68 defplist postTuple plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|@Tuple| '|postTran|) '|postTuple|))
```

———-

7.2.69 defun postTuple

[postTranList p339]

— defun postTuple —

```
(defun |postTuple| (arg)
 (cond
  ((and (consp arg) (eq (qrest arg) nil) (eq (qfirst arg) '|@Tuple|))
   arg)
  ((and (consp arg) (eq (qfirst arg) '|@Tuple|) (consp (qrest arg)))
   (cons '|@Tuple| (postTranList (cdr arg))))))
```

———-

7.2.70 defplist postTupleCollect plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|TupleCollect| '|postTran|) '|postTupleCollect|))
```

———-

7.2.71 defun postTupleCollect

[postCollect p349]

— defun postTupleCollect —

```
(defun |postTupleCollect| (arg)
 (let (constructOp tmp1 x m)
  (setq constructOp (car arg))
  (setq tmp1 (reverse (cdr arg)))
  (setq x (car tmp1))
  (setq m (nreverse (cdr tmp1)))
  (|postCollect| (cons constructOp (append m (list (list '|construct| x)))))))
```

———-

7.2.72 defplist postWhere plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|where| '|postTran|) '|postWhere|))
```

———-

7.2.73 defun postWhere

[postTran p338]
[postTranList p339]

— defun postWhere —

```
(defun |postWhere| (arg)
 (let (b x)
  (setq b (third arg))
  (setq x (if (and (consp b) (eq (qfirst b) '|Block|)) (qrest b) (list b)))
  (cons '|where| (cons (|postTran| (second arg)) (postTranList x)))))
```

———-

7.2.74 defplist postWith plist

— postvars —

```
(eval-when (eval load)
 (setf (get '|with| '|postTran|) '|postWith|))
```

———-

7.2.75 defun postWith

[postTran p338]
[$insidePostCategoryIfTrue p??]

— defun postWith —

```
(defun |postWith| (arg)
 (let (|$insidePostCategoryIfTrue| a)
 (declare (special |$insidePostCategoryIfTrue|))
  (setq |$insidePostCategoryIfTrue| t)
  (setq a (|postTran| (second arg)))
  (cond
   ((and (consp a) (member (qfirst a) '(signature attribute if)))
     (list 'category a))
   ((and (consp a) (eq (qfirst a) 'progn))
     (cons 'category (qrest a)))
   (t a))))
```

———-

7.3 Support routines

7.3.1 defun setDefOp

[$defOp p??]
[$topOp p??]

— defun setDefOp —

```
(defun setDefOp (f)
 (let (tmp1)
  (declare (special |$defOp| |$topOp|))
  (when (and (consp f) (eq (qfirst f) '|:|)
             (consp (setq tmp1 (qrest f))))
    (setq f (qfirst tmp1)))
  (unless (atom f) (setq f (car f)))
  (if |$topOp|
   (setq |$defOp| f)
   (setq |$topOp| f))))
```

———

7.3.2 defun aplTran

[aplTran1 p369]
[containsBang p371]
[$genno p??]
[$boot p??]

— defun aplTran —

```
(defun aplTran (x)
 (let ($genno u)
 (declare (special $genno $boot))
  (cond
   ($boot x)
   (t
    (setq $genno 0)
    (setq u (aplTran1 x))
    (cond
     ((containsBang u)
      (|throwKeyedMsg|
       (format nil
         " AXIOM cannot now process ! in the way you have used it. ~
          Use parentheses, if appropriate.")
       nil))
     (t u))))))
```

———

7.3.3 defun aplTran1

[aplTranList p370]
[aplTran1 p369]
[hasAplExtension p370]
[nreverse0 p??]
[$boot p??]

— **defun aplTran1** —

```
(defun aplTran1 (x)
 (let (op argl1 argl f y opprime yprime tmp1 arglAssoc futureArgl g)
 (declare (special $boot))
  (if (atom x)
   x
   (progn
    (setq op (car x))
    (setq argl1 (cdr x))
    (setq argl (aplTranList argl1))
    (cond
     ((eq op '!)
      (cond
       ((and (consp argl)
             (progn
              (setq f (qfirst argl))
              (setq tmp1 (qrest argl))
              (and (consp tmp1)
                   (eq (qrest tmp1) nil)
                   (progn
                    (setq y (qfirst tmp1))
                    t))))
         (cond
          ((and (consp y)
                (progn
                 (setq opprime (qfirst y))
                 (setq yprime (qrest y))
                 t)
                (eq opprime '!))
            (aplTran1 (cons op (cons op (cons f yprime)))))
          ($boot
           (cons 'collect
            (cons
             (list 'in (setq g (genvar)) (aplTran1 y))
              (list (list f g ) ))))
          (t
           (list 'map f (aplTran1 y) ))))
       (t x)))
     ((progn
       (setq tmp1 (hasAplExtension argl))
       (and (consp tmp1)
            (progn
             (setq arglAssoc (qfirst tmp1))
             (setq futureArgl (qrest tmp1))
             t)))
```

```
       (cons '|reshape|
        (cons
         (cons 'collect
          (append
           (do ((tmp3 arglAssoc (cdr tmp3)) (tmp4 nil))
               ((or (atom tmp3)
                    (progn (setq tmp4 (car tmp3)) nil)
                    (progn
                      (setq g (car tmp4))
                      (setq a (cdr tmp4))
                      nil))
                   (nreverse0 tmp2))
              (push (list 'in g (list 'identity a))) tmp2))
          (list (aplTran1 (cons op futureArgl)))))
         (list (cdar arglAssoc))))
     (t (cons op argl)))))))
```

——–

7.3.4 defun aplTranList

[aplTran1 p369]
[aplTranList p370]

— defun aplTranList —

```
(defun aplTranList (x)
 (if (atom x)
  x
  (cons (aplTran1 (car x)) (aplTranList (cdr x)))))
```

——–

7.3.5 defun hasAplExtension

[nreverse0 p??]
[deepestExpression p371]
[genvar p??]
[aplTran1 p369]

— defun hasAplExtension —

```
(defun hasAplExtension (argl)
 (let (tmp2 tmp3 y z g arglAssoc u)
  (when
   (dolist (x argl tmp2)
      (setq tmp2 (or tmp2 (and (consp x) (eq (qfirst x) '!)))))
    (setq u
     (dolist (x argl (nreverse0 tmp3))
      (push
        (if (and (consp x) (eq (qfirst x) '!)
                 (consp (qrest x)) (eq (qcddr x) nil))
```

```
         (progn
           (setq y (qsecond x))
           (setq z (deepestExpression y))
           (setq arglAssoc
            (cons (cons (setq g (genvar)) (aplTran1 z)) arglAssoc))
           (subst g z y :test #'equal))
          x)
         tmp3)))
    (cons arglAssoc u))))
```

———-

7.3.6 defun deepestExpression

[deepestExpression p371]

— defun deepestExpression —

```
(defun deepestExpression (x)
 (if (and (consp x) (eq (qfirst x) '!)
          (consp (qrest x)) (eq (qcddr x) nil))
   (deepestExpression (qsecond x))
   x))
```

———-

7.3.7 defun containsBang

[containsBang p371]

— defun containsBang —

```
(defun containsBang (u)
 (let (tmp2)
  (cond
   ((atom u) (eq u '!))
   ((and (consp u) (equal (qfirst u) 'quote)
         (consp (qrest u)) (eq (qcddr u) nil))
     nil)
   (t
     (dolist (x u tmp2)
       (setq tmp2 (or tmp2 (containsBang x))))))))
```

———-

7.3.8 defun getScriptName

[getScriptName identp (vol5)]
[postError p341]
[internl p**??**]

[decodeScripts p372]
[getScriptName pname (vol5)]

— defun getScriptName —

```
(defun getScriptName (op a numberOfFunctionalArgs)
 (when (null (identp op))
   (postError (list "   " op " cannot have scripts" )))
 (internl '* (princ-to-string numberOfFunctionalArgs)
             (decodeScripts a) (pname op)))
```

———-

7.3.9 defun decodeScripts

[strconc p??]
[decodeScripts p372]

— defun decodeScripts —

```
(defun decodeScripts (a)
 (labels (
  (fn (a)
   (let ((tmp1 0))
    (if (and (consp a) (eq (qfirst a) '|,|))
     (dolist (x (qrest a) tmp1) (setq tmp1 (+ tmp1 (fn x))))
     1))))
  (cond
   ((and (consp a) (eq (qfirst a) '|PrefixSC|)
         (consp (qrest a)) (eq (qcddr a) nil))
    (strconc (princ-to-string 0) (decodeScripts (qsecond a))))
   ((and (consp a) (eq (qfirst a) '|;|))
    (apply 'strconc (loop for x in (qrest a) collect (decodeScripts x))))
   ((and (consp a) (eq (qfirst a) '|,|))
    (princ-to-string (fn a)))
   (t
    (princ-to-string 1)))))
```

———-

Chapter 8

DEF forms

8.0.10 defvar $defstack

— **initvars** —

```
(defvar $defstack nil)
```

———-

8.0.11 defvar $is-spill

— **initvars** —

```
(defvar $is-spill nil)
```

———-

8.0.12 defvar $is-spill-list

— **initvars** —

```
(defvar $is-spill-list nil)
```

———-

8.0.13 defvar $vl

— **initvars** —

```
(defvar $vl nil)
```

———-

8.0.14 defvar $is-gensymlist

— initvars —

```
(defvar $is-gensymlist nil)
```

———-

8.0.15 defvar initial-gensym

— initvars —

```
(defvar initial-gensym (list (gensym)))
```

———-

8.0.16 defvar $is-eqlist

— initvars —

```
(defvar $is-eqlist nil)
```

———-

8.0.17 defun hackforis

[hackforis1 p374]

— defun hackforis —

```
(defun hackforis (l) (mapcar #'hackforis1 L))
```

———-

8.0.18 defun hackforis1

[eqcar p??]

— defun hackforis1 —

```
(defun hackforis1 (x)
 (if (and (member (ifcar x) '(in on)) (eqcar (second x) 'is))
  (cons (first x) (cons (cons 'setq (cdadr x)) (cddr x)))
  x))
```

———-

8.0.19 defun unTuple

— defun unTuple —

```
(defun |unTuple| (x)
 (if (and (consp x) (eq (qfirst x) '|@Tuple|))
  (qrest x)
  (list x)))
```

———-

8.1 The PARSE code

8.1.1 defvar tmptok

— initvars —

```
(defvar |tmptok| nil)
```

———-

8.1.2 defvar tok

— initvars —

```
(defvar tok nil)
```

———-

8.1.3 defvar ParseMode

— initvars —

```
(defvar |ParseMode| nil)
```

———-

8.1.4 defvar definition-name

— initvars —

```
(defvar definition-name nil)
```

———-

8.1.5 defvar lablasoc

— initvars —

```
(defvar lablasoc nil)
```

———-

8.1.6 defun PARSE-NewExpr

[match-string p408]
[action p419]
[PARSE-NewExpr processSynonyms (vol5)]
[must p419]
[current-symbol p414]
[PARSE-Statement p380]
[definition-name p375]

— defun PARSE-NewExpr —

```
(defun |PARSE-NewExpr| ()
 (or (and (match-string ")") (action (|processSynonyms|))
          (must (|PARSE-Command|)))
     (and (action (setq definition-name (current-symbol)))
          (|PARSE-Statement|))))
```

———-

8.1.7 defun PARSE-Command

[match-advance-string p409]
[must p419]
[PARSE-SpecialKeyWord p377]
[PARSE-SpecialCommand p377]
[push-reduction p421]

— defun PARSE-Command —

```
(defun |PARSE-Command| ()
 (and (match-advance-string ")") (must (|PARSE-SpecialKeyWord|))
      (must (|PARSE-SpecialCommand|))
      (push-reduction '|PARSE-Command| nil)))
```

———-

8.1.8 defun PARSE-SpecialKeyWord

[match-current-token p413]
[action p419]
[token-symbol p??]
[current-token p414]
[PARSE-SpecialKeyWord unAbbreviateKeyword (vol5)]
[current-symbol p414]

— defun PARSE-SpecialKeyWord —

```
(defun |PARSE-SpecialKeyWord| ()
 (and (match-current-token 'identifier)
      (action (setf (token-symbol (current-token))
                    (|unAbbreviateKeyword| (current-symbol))))))
```

———-

8.1.9 defun PARSE-SpecialCommand

[match-advance-string p409]
[bang p??]
[optional p419]
[PARSE-Expression p382]
[push-reduction p421]
[PARSE-SpecialCommand p377]
[pop-stack-1 p495]
[PARSE-CommandTail p379]
[must p419]
[current-symbol p414]
[action p419]
[PARSE-TokenList p379]
[PARSE-TokenCommandTail p378]
[star p420]
[PARSE-PrimaryOrQM p379]
[PARSE-CommandTail p379]
[$noParseCommands p??]
[$tokenCommands p??]

— defun PARSE-SpecialCommand —

```
(defun |PARSE-SpecialCommand| ()
 (declare (special |$noParseCommands| |$tokenCommands|))
 (or (and (match-advance-string "show")
          (bang fil_test
                (optional
                    (or (match-advance-string "?")
                        (|PARSE-Expression|))))
          (push-reduction '|PARSE-SpecialCommand|
              (list '|show| (pop-stack-1)))
          (must (|PARSE-CommandTail|)))
```

```
    (and (member (current-symbol) |$noParseCommands|)
         (action (funcall (current-symbol))))
    (and (member (current-symbol) |$tokenCommands|)
         (|PARSE-TokenList|) (must (|PARSE-TokenCommandTail|)))
    (and (star repeator (|PARSE-PrimaryOrQM|))
         (must (|PARSE-CommandTail|)))))
```

———-

8.1.10 defun PARSE-TokenCommandTail

[bang p??]
[optional p419]
[star p420]
[PARSE-TokenOption p378]
[atEndOfLine p??]
[push-reduction p421]
[PARSE-TokenCommandTail p378]
[pop-stack-2 p495]
[pop-stack-1 p495]
[action p419]
[PARSE-TokenCommandTail systemCommand (vol5)]

— defun PARSE-TokenCommandTail —

```
(defun |PARSE-TokenCommandTail| ()
 (and (bang fil_test (optional (star repeator (|PARSE-TokenOption|))))
      (|atEndOfLine|)
      (push-reduction '|PARSE-TokenCommandTail|
          (cons (pop-stack-2) (append (pop-stack-1) nil)))
      (action (|systemCommand| (pop-stack-1)))))
```

———-

8.1.11 defun PARSE-TokenOption

[match-advance-string p409]
[must p419]
[PARSE-TokenList p379]

— defun PARSE-TokenOption —

```
(defun |PARSE-TokenOption| ()
 (and (match-advance-string ")") (must (|PARSE-TokenList|))))
```

———-

8.1.12 defun PARSE-TokenList

[star p420]
[isTokenDelimiter p411]
[push-reduction p421]
[current-symbol p414]
[action p419]
[advance-token p415]

— defun PARSE-TokenList —

```
(defun |PARSE-TokenList| ()
 (star repeator
  (and (not (|isTokenDelimiter|))
       (push-reduction '|PARSE-TokenList| (current-symbol))
       (action (advance-token)))))
```

———-

8.1.13 defun PARSE-CommandTail

[bang p??]
[optional p419]
[star p420]
[push-reduction p421]
[PARSE-Option p380]
[PARSE-CommandTail p379]
[pop-stack-2 p495]
[pop-stack-1 p495]
[action p419]
[PARSE-CommandTail systemCommand (vol5)]

— defun PARSE-CommandTail —

```
(defun |PARSE-CommandTail| ()
 (and (bang fil_test (optional (star repeator (|PARSE-Option|))))
      (|atEndOfLine|)
      (push-reduction '|PARSE-CommandTail|
          (cons (pop-stack-2) (append (pop-stack-1) nil)))
      (action (|systemCommand| (pop-stack-1)))))
```

———-

8.1.14 defun PARSE-PrimaryOrQM

[match-advance-string p409]
[push-reduction p421]
[PARSE-PrimaryOrQM p379]
[PARSE-Primary p391]

— **defun PARSE-PrimaryOrQM** —

```
(defun |PARSE-PrimaryOrQM| ()
 (or (and (match-advance-string "?")
          (push-reduction '|PARSE-PrimaryOrQM| '?))
     (|PARSE-Primary|)))
```

———

8.1.15 defun PARSE-Option

[match-advance-string p409]
[must p419]
[star p420]
[PARSE-PrimaryOrQM p379]

— **defun PARSE-Option** —

```
(defun |PARSE-Option| ()
 (and (match-advance-string ")")
      (must (star repeator (|PARSE-PrimaryOrQM|)))))
```

———

8.1.16 defun PARSE-Statement

[PARSE-Expr p383]
[optional p419]
[star p420]
[match-advance-string p409]
[must p419]
[push-reduction p421]
[pop-stack-2 p495]
[pop-stack-1 p495]

— **defun PARSE-Statement** —

```
(defun |PARSE-Statement| ()
  (and (|PARSE-Expr| 0)
       (optional
           (and (star repeator
                      (and (match-advance-string ",")
                           (must (|PARSE-Expr| 0))))
                (push-reduction '|PARSE-Statement|
                    (cons '|Series|
                          (cons (pop-stack-2)
                                (append (pop-stack-1) nil)))))))))
```

———

8.1.17 defun PARSE-InfixWith

[PARSE-With p381]
[push-reduction p421]
[pop-stack-2 p495]
[pop-stack-1 p495]

— defun PARSE-InfixWith —

```
(defun |PARSE-InfixWith| ()
  (and (|PARSE-With|)
       (push-reduction '|PARSE-InfixWith|
           (list '|Join| (pop-stack-2) (pop-stack-1)))))
```

———-

8.1.18 defun PARSE-With

[match-advance-string p409]
[must p419]
[push-reduction p421]
[pop-stack-1 p495]

— defun PARSE-With —

```
(defun |PARSE-With| ()
  (and (match-advance-string "with") (must (|PARSE-Category|))
       (push-reduction '|PARSE-With|
           (cons '|with| (cons (pop-stack-1) nil)))))
```

———-

8.1.19 defun PARSE-Category

[match-advance-string p409]
[must p419]
[bang p??]
[optional p419]
[push-reduction p421]
[PARSE-Expression p382]
[PARSE-Category p381]
[pop-stack-3 p495]
[pop-stack-2 p495]
[pop-stack-1 p495]
[star p420]
[line-number p??]
[PARSE-Application p389]
[action p419]
[recordSignatureDocumentation p424]
[nth-stack p496]

[recordAttributeDocumentation p424]
[PARSE-Category current-line (vol5)]

— defun PARSE-Category —

```
(defun |PARSE-Category| ()
 (let (g1)
  (or (and (match-advance-string "if") (must (|PARSE-Expression|))
           (must (match-advance-string "then"))
           (must (|PARSE-Category|))
           (bang fil_test
                 (optional
                     (and (match-advance-string "else")
                          (must (|PARSE-Category|)))))
           (push-reduction '|PARSE-Category|
             (list '|if| (pop-stack-3) (pop-stack-2) (pop-stack-1))))
      (and (match-advance-string "(") (must (|PARSE-Category|))
           (bang fil_test
                 (optional
                     (star repeator
                           (and (match-advance-string ";")
                                (must (|PARSE-Category|))))))
           (must (match-advance-string ")"))
           (push-reduction '|PARSE-Category|
               (cons 'category
                     (cons (pop-stack-2)
                           (append (pop-stack-1) nil)))))
      (and (action (setq g1 (line-number current-line)))
           (|PARSE-Application|)
           (must (or (and (match-advance-string ":")
                          (must (|PARSE-Expression|))
                          (push-reduction '|PARSE-Category|
                           (list '|Signature| (pop-stack-2) (pop-stack-1) ))
                          (action (|recordSignatureDocumentation|
                                   (nth-stack 1) g1)))
                     (and (push-reduction '|PARSE-Category|
                            (list '|Attribute| (pop-stack-1) ))
                          (action (|recordAttributeDocumentation|
                                   (nth-stack 1) g1))))))))
```

———

8.1.20 defun PARSE-Expression

[PARSE-Expr p383]
[PARSE-rightBindingPowerOf p385]
[make-symbol-of p414]
[push-reduction p421]
[pop-stack-1 p495]
[ParseMode p375]
[prior-token p95]

— defun PARSE-Expression —

```
(defun |PARSE-Expression| ()
 (declare (special prior-token))
 (and (|PARSE-Expr|
          (|PARSE-rightBindingPowerOf| (make-symbol-of prior-token)
              |ParseMode|))
      (push-reduction '|PARSE-Expression| (pop-stack-1))))
```

———-

8.1.21 defun PARSE-Import

[match-advance-string p409]
[must p419]
[PARSE-Expr p383]
[bang p??]
[optional p419]
[star p420]
[push-reduction p421]
[pop-stack-2 p495]
[pop-stack-1 p495]

— defun PARSE-Import —

```
(defun |PARSE-Import| ()
  (and (match-advance-string "import") (must (|PARSE-Expr| 1000))
       (bang fil_test
             (optional
                 (star repeator
                       (and (match-advance-string ",")
                            (must (|PARSE-Expr| 1000))))))
       (push-reduction '|PARSE-Import|
           (cons '|import|
                 (cons (pop-stack-2) (append (pop-stack-1) nil))))))
```

———-

8.1.22 defun PARSE-Expr

[PARSE-NudPart p384]
[PARSE-LedPart p384]
[optional p419]
[star p420]
[push-reduction p421]
[pop-stack-1 p495]

— defun PARSE-Expr —

```
(defun |PARSE-Expr| (rbp)
 (declare (special rbp))
```

```
 (and (|PARSE-NudPart| rbp)
      (optional (star opt_expr (|PARSE-LedPart| rbp)))
      (push-reduction '|PARSE-Expr| (pop-stack-1))))
```

———-

8.1.23 defun PARSE-LedPart

[PARSE-Operation p384]
[push-reduction p421]
[pop-stack-1 p495]

— defun PARSE-LedPart —

```
(defun |PARSE-LedPart| (rbp)
  (declare (special rbp))
  (and (|PARSE-Operation| '|Led| rbp)
       (push-reduction '|PARSE-LedPart| (pop-stack-1))))
```

———-

8.1.24 defun PARSE-NudPart

[PARSE-Operation p384]
[PARSE-Reduction p388]
[PARSE-Form p388]
[push-reduction p421]
[pop-stack-1 p495]
[rbp p**??**]

— defun PARSE-NudPart —

```
(defun |PARSE-NudPart| (rbp)
  (declare (special rbp))
  (and (or (|PARSE-Operation| '|Nud| rbp) (|PARSE-Reduction|)
           (|PARSE-Form|))
       (push-reduction '|PARSE-NudPart| (pop-stack-1))))
```

———-

8.1.25 defun PARSE-Operation

[match-current-token p413]
[current-symbol p414]
[PARSE-leftBindingPowerOf p385]
[lt p**??**]
[getl p**??**]
[action p419]
[PARSE-rightBindingPowerOf p385]

[PARSE-getSemanticForm p385]
[elemn p??]
[ParseMode p375]
[rbp p??]
[tmptok p375]

— defun PARSE-Operation —

```
(defun |PARSE-Operation| (|ParseMode| rbp)
 (declare (special |ParseMode| rbp |tmptok|))
 (and (not (match-current-token 'identifier))
      (getl (setq |tmptok| (current-symbol)) |ParseMode|)
      (lt rbp (|PARSE-leftBindingPowerOf| |tmptok| |ParseMode|))
      (action (setq rbp (|PARSE-rightBindingPowerOf| |tmptok| |ParseMode|)))
      (|PARSE-getSemanticForm| |tmptok| |ParseMode|
          (elemn (getl |tmptok| |ParseMode|) 5 nil))))
```

———

8.1.26 defun PARSE-leftBindingPowerOf

[getl p??]
[elemn p??]

— defun PARSE-leftBindingPowerOf —

```
(defun |PARSE-leftBindingPowerOf| (x ind)
 (declare (special x ind))
 (let ((y (getl x ind))) (if y (elemn y 3 0) 0)))
```

———

8.1.27 defun PARSE-rightBindingPowerOf

[getl p??]
[elemn p??]

— defun PARSE-rightBindingPowerOf —

```
(defun |PARSE-rightBindingPowerOf| (x ind)
  (declare (special x ind))
  (let ((y (getl x ind))) (if y (elemn y 4 105) 105)))
```

———

8.1.28 defun PARSE-getSemanticForm

[PARSE-Prefix p386]
[PARSE-Infix p386]

— defun PARSE-getSemanticForm —

```
(defun |PARSE-getSemanticForm| (x ind y)
  (declare (special x ind y))
  (or (and y (eval y)) (and (eq ind '|Nud|) (|PARSE-Prefix|))
      (and (eq ind '|Led|) (|PARSE-Infix|))))
```

———-

8.1.29 defun PARSE-Prefix

[push-reduction p421]
[current-symbol p414]
[action p419]
[advance-token p415]
[optional p419]
[PARSE-TokTail p387]
[must p419]
[PARSE-Expression p382]
[push-reduction p421]
[pop-stack-2 p495]
[pop-stack-1 p495]

— defun PARSE-Prefix —

```
(defun |PARSE-Prefix| ()
  (and (push-reduction '|PARSE-Prefix| (current-symbol))
       (action (advance-token)) (optional (|PARSE-TokTail|))
       (must (|PARSE-Expression|))
       (push-reduction '|PARSE-Prefix|
           (list (pop-stack-2) (pop-stack-1)))))
```

———-

8.1.30 defun PARSE-Infix

[push-reduction p421]
[current-symbol p414]
[action p419]
[advance-token p415]
[optional p419]
[PARSE-TokTail p387]
[must p419]
[PARSE-Expression p382]
[pop-stack-2 p495]
[pop-stack-1 p495]

— defun PARSE-Infix —

```
(defun |PARSE-Infix| ()
```

```
  (and (push-reduction '|PARSE-Infix| (current-symbol))
       (action (advance-token)) (optional (|PARSE-TokTail|))
       (must (|PARSE-Expression|))
       (push-reduction '|PARSE-Infix|
           (list (pop-stack-2) (pop-stack-2) (pop-stack-1) ))))
```

———-

8.1.31 defun PARSE-TokTail

[current-symbol p414]
[current-char p416]
[char-eq p417]
[copy-token p??]
[action p419]
[PARSE-Qualification p387]
[$boot p??]

— defun PARSE-TokTail —

```
(defun |PARSE-TokTail| ()
 (let (g1)
  (and (null $boot) (eq (current-symbol) '$)
       (or (alpha-char-p (current-char))
           (char-eq (current-char) "$")
           (char-eq (current-char) "%")
           (char-eq (current-char) "("))
       (action (setq g1 (copy-token prior-token)))
       (|PARSE-Qualification|) (action (setq prior-token g1)))))
```

———-

8.1.32 defun PARSE-Qualification

[match-advance-string p409]
[must p419]
[PARSE-Primary1 p391]
[push-reduction p421]
[dollarTran p418]
[pop-stack-1 p495]

— defun PARSE-Qualification —

```
(defun |PARSE-Qualification| ()
  (and (match-advance-string "$") (must (|PARSE-Primary1|))
       (push-reduction '|PARSE-Qualification|
           (|dollarTran| (pop-stack-1) (pop-stack-1)))))
```

———-

8.1.33 defun PARSE-Reduction

[PARSE-ReductionOp p388]
[must p419]
[PARSE-Expr p383]
[push-reduction p421]
[pop-stack-2 p495]
[pop-stack-1 p495]

— defun PARSE-Reduction —

```
(defun |PARSE-Reduction| ()
 (and (|PARSE-ReductionOp|) (must (|PARSE-Expr| 1000))
      (push-reduction '|PARSE-Reduction|
          (list '|Reduce| (pop-stack-2) (pop-stack-1) ))))
```

———-

8.1.34 defun PARSE-ReductionOp

[getl p??]
[current-symbol p414]
[match-next-token p413]
[action p419]
[advance-token p415]

— defun PARSE-ReductionOp —

```
(defun |PARSE-ReductionOp| ()
  (and (getl (current-symbol) '|Led|)
       (match-next-token 'special-char (code-char 47))
       (push-reduction '|PARSE-ReductionOp| (current-symbol))
       (action (advance-token)) (action (advance-token))))
```

———-

8.1.35 defun PARSE-Form

[match-advance-string p409]
[bang p??]
[optional p419]
[must p419]
[push-reduction p421]
[pop-stack-1 p495]
[PARSE-Application p389]

— defun PARSE-Form —

```
(defun |PARSE-Form| ()
  (or (and (match-advance-string "iterate")
           (bang fil_test
```

```
              (optional
                  (and (match-advance-string "from")
                       (must (|PARSE-Label|))
                       (push-reduction '|PARSE-Form|
                          (list (pop-stack-1))))))
        (push-reduction '|PARSE-Form|
            (cons '|iterate| (append (pop-stack-1) nil))))
   (and (match-advance-string "yield") (must (|PARSE-Application|))
        (push-reduction '|PARSE-Form|
            (list '|yield| (pop-stack-1))))
   (|PARSE-Application|)))
```

———-

8.1.36 defun PARSE-Application

[PARSE-Primary p391]
[optional p419]
[star p420]
[PARSE-Selector p390]
[PARSE-Application p389]
[push-reduction p421]
[pop-stack-2 p495]
[pop-stack-1 p495]

— defun PARSE-Application —

```
(defun |PARSE-Application| ()
 (and (|PARSE-Primary|) (optional (star opt_expr (|PARSE-Selector|)))
      (optional
          (and (|PARSE-Application|)
               (push-reduction '|PARSE-Application|
                   (list (pop-stack-2) (pop-stack-1)))))))
```

———-

8.1.37 defun PARSE-Label

[match-advance-string p409]
[must p419]
[PARSE-Name p397]

— defun PARSE-Label —

```
(defun |PARSE-Label| ()
  (and (match-advance-string "<<") (must (|PARSE-Name|))
       (must (match-advance-string ">>"))))
```

———-

8.1.38 defun PARSE-Selector

[current-symbol p414]
[char-ne p417]
[current-char p416]
[match-advance-string p409]
[must p419]
[PARSE-PrimaryNoFloat p390]
[push-reduction p421]
[pop-stack-2 p495]
[pop-stack-1 p495]
[PARSE-Float p392]
[PARSE-Primary p391]
[$boot p??]

— defun PARSE-Selector —

```
(defun |PARSE-Selector| ()
  (declare (special $boot))
  (or (and nonblank (eq (current-symbol) '|.|)
           (char-ne (current-char) '| |) (match-advance-string ".")
           (must (|PARSE-PrimaryNoFloat|))
           (must (or (and $boot
                          (push-reduction '|PARSE-Selector|
                              (list 'elt (pop-stack-2) (pop-stack-1))))
                     (push-reduction '|PARSE-Selector|
                         (list (pop-stack-2) (pop-stack-1))))))
      (and (or (|PARSE-Float|)
               (and (match-advance-string ".")
                    (must (|PARSE-Primary|))))
           (must (or (and $boot
                          (push-reduction '|PARSE-Selector|
                              (list 'elt (pop-stack-2) (pop-stack-1))))
                     (push-reduction '|PARSE-Selector|
                         (list (pop-stack-2) (pop-stack-1))))))))
```

———-

8.1.39 defun PARSE-PrimaryNoFloat

[PARSE-Primary1 p391]
[optional p419]
[PARSE-TokTail p387]

— defun PARSE-PrimaryNoFloat —

```
(defun |PARSE-PrimaryNoFloat| ()
  (and (|PARSE-Primary1|) (optional (|PARSE-TokTail|))))
```

———-

8.1.40 defun PARSE-Primary

[PARSE-Float p392]
[PARSE-PrimaryNoFloat p390]

— defun PARSE-Primary —

```
(defun |PARSE-Primary| ()
  (or (|PARSE-Float|) (|PARSE-PrimaryNoFloat|)))
```

———-

8.1.41 defun PARSE-Primary1

[PARSE-VarForm p396]
[optional p419]
[current-symbol p414]
[PARSE-Primary1 p391]
[must p419]
[pop-stack-2 p495]
[pop-stack-1 p495]
[push-reduction p421]
[PARSE-Quad p395]
[PARSE-String p395]
[PARSE-IntegerTok p394]
[PARSE-FormalParameter p395]
[match-string p408]
[PARSE-Data p397]
[match-advance-string p409]
[PARSE-Expr p383]
[PARSE-Sequence p400]
[PARSE-Enclosure p394]
[$boot p??]

— defun PARSE-Primary1 —

```
(defun |PARSE-Primary1| ()
 (declare (special $boot))
  (or (and (|PARSE-VarForm|)
           (optional
               (and nonblank (eq (current-symbol) '|(|)
                    (must (|PARSE-Primary1|))
                    (push-reduction '|PARSE-Primary1|
                        (list (pop-stack-2) (pop-stack-1))))))
      (|PARSE-Quad|) (|PARSE-String|) (|PARSE-IntegerTok|)
      (|PARSE-FormalParameter|)
      (and (match-string "'")
           (must (or (and $boot (|PARSE-Data|))
                     (and (match-advance-string "'")
                          (must (|PARSE-Expr| 999))
                          (push-reduction '|PARSE-Primary1|
                              (list 'quote (pop-stack-1)))))))
```

```
    (|PARSE-Sequence|) (|PARSE-Enclosure|)))
```

———-

8.1.42 defun PARSE-Float

[PARSE-FloatBase p392]
[must p419]
[PARSE-FloatExponent p393]
[push-reduction p421]
[make-float p??]
[pop-stack-4 p496]
[pop-stack-3 p495]
[pop-stack-2 p495]
[pop-stack-1 p495]

— defun PARSE-Float —

```
(defun |PARSE-Float| ()
  (and (|PARSE-FloatBase|)
       (must (or (and nonblank (|PARSE-FloatExponent|))
                 (push-reduction '|PARSE-Float| 0)))
       (push-reduction '|PARSE-Float|
           (make-float (pop-stack-4) (pop-stack-2) (pop-stack-2)
               (pop-stack-1)))))
```

———-

8.1.43 defun PARSE-FloatBase

[current-symbol p414]
[char-eq p417]
[current-char p416]
[char-ne p417]
[next-char p416]
[PARSE-IntegerTok p394]
[must p419]
[PARSE-FloatBasePart p393]
[PARSE-IntegerTok p394]
[push-reduction p421]
[PARSE-FloatBase digitp (vol5)]

— defun PARSE-FloatBase —

```
(defun |PARSE-FloatBase| ()
  (or (and (integerp (current-symbol)) (char-eq (current-char) ".")
           (char-ne (next-char) ".") (|PARSE-IntegerTok|)
           (must (|PARSE-FloatBasePart|)))
      (and (integerp (current-symbol))
           (char-eq (char-upcase (current-char)) 'e)
```

```
           (|PARSE-IntegerTok|) (push-reduction '|PARSE-FloatBase| 0)
           (push-reduction '|PARSE-FloatBase| 0))
      (and (digitp (current-char)) (eq (current-symbol) '|.|)
           (push-reduction '|PARSE-FloatBase| 0)
           (|PARSE-FloatBasePart|))))
```

———-

8.1.44 defun PARSE-FloatBasePart

[match-advance-string p409]
[must p419]
[PARSE-FloatBasePart digitp (vol5)]
[current-char p416]
[push-reduction p421]
[token-nonblank p??]
[current-token p414]
[PARSE-IntegerTok p394]

— defun PARSE-FloatBasePart —

```
(defun |PARSE-FloatBasePart| ()
  (and (match-advance-string ".")
       (must (or (and (digitp (current-char))
                      (push-reduction '|PARSE-FloatBasePart|
                          (token-nonblank (current-token)))
                      (|PARSE-IntegerTok|))
                 (and (push-reduction '|PARSE-FloatBasePart| 0)
                      (push-reduction '|PARSE-FloatBasePart| 0))))))
```

———-

8.1.45 defun PARSE-FloatExponent

[current-symbol p414]
[current-char p416]
[action p419]
[advance-token p415]
[PARSE-IntegerTok p394]
[match-advance-string p409]
[must p419]
[push-reduction p421]
[PARSE-FloatExponent identp (vol5)]
[floatexpid p418]

— defun PARSE-FloatExponent —

```
(defun |PARSE-FloatExponent| ()
 (let (g1)
  (or (and (member (current-symbol) '(e |e|))
```

```
        (find (current-char) "+-") (action (advance-token))
        (must (or (|PARSE-IntegerTok|)
                  (and (match-advance-string "+")
                       (must (|PARSE-IntegerTok|)))
                  (and (match-advance-string "-")
                       (must (|PARSE-IntegerTok|))
                       (push-reduction '|PARSE-FloatExponent|
                           (- (pop-stack-1))))
                  (push-reduction '|PARSE-FloatExponent| 0))))
   (and (identp (current-symbol))
        (setq g1 (floatexpid (current-symbol)))
        (action (advance-token))
        (push-reduction '|PARSE-FloatExponent| g1)))))
```

———-

8.1.46 defun PARSE-Enclosure

[match-advance-string p409]
[must p419]
[PARSE-Expr p383]
[push-reduction p421]
[pop-stack-1 p495]

— defun PARSE-Enclosure —

```
(defun |PARSE-Enclosure| ()
  (or (and (match-advance-string "(")
        (must (or (and (|PARSE-Expr| 6)
                       (must (match-advance-string ")")))
                  (and (match-advance-string ")")
                       (push-reduction '|PARSE-Enclosure|
                           (list '|@Tuple|))))))
   (and (match-advance-string "{")
        (must (or (and (|PARSE-Expr| 6)
                       (must (match-advance-string "}"))
                       (push-reduction '|PARSE-Enclosure|
                           (cons '|brace|
                            (list (list '|construct| (pop-stack-1))))))
                  (and (match-advance-string "}")
                       (push-reduction '|PARSE-Enclosure|
                           (list '|brace|))))))))
```

———-

8.1.47 defun PARSE-IntegerTok

[parse-number p493]

— defun PARSE-IntegerTok —

```
(defun |PARSE-IntegerTok| () (parse-number))
```

———-

8.1.48 defun PARSE-FormalParameter

[PARSE-FormalParameterTok p395]

— defun PARSE-FormalParameter —

```
(defun |PARSE-FormalParameter| () (|PARSE-FormalParameterTok|))
```

———-

8.1.49 defun PARSE-FormalParameterTok

[parse-argument-designator p493]

— defun PARSE-FormalParameterTok —

```
(defun |PARSE-FormalParameterTok| () (parse-argument-designator))
```

———-

8.1.50 defun PARSE-Quad

[match-advance-string p409]
[push-reduction p421]
[PARSE-GliphTok p399]
[$boot p**??**]

— defun PARSE-Quad —

```
(defun |PARSE-Quad| ()
 (or (and (match-advance-string "$")
          (push-reduction '|PARSE-Quad| '$))
     (and $boot (|PARSE-GliphTok| '|.|)
          (push-reduction '|PARSE-Quad| '|.|))))
```

———-

8.1.51 defun PARSE-String

[parse-spadstring p492]

— defun PARSE-String —

```
(defun |PARSE-String| () (parse-spadstring))
```

———-

8.1.52 defun PARSE-VarForm

[PARSE-Name p397]
[optional p419]
[PARSE-Scripts p396]
[push-reduction p421]
[pop-stack-2 p495]
[pop-stack-1 p495]

— defun PARSE-VarForm —

```
(defun |PARSE-VarForm| ()
  (and (|PARSE-Name|)
       (optional
           (and (|PARSE-Scripts|)
                (push-reduction '|PARSE-VarForm|
                    (list '|Scripts| (pop-stack-2) (pop-stack-1)))))
       (push-reduction '|PARSE-VarForm| (pop-stack-1))))
```

———-

8.1.53 defun PARSE-Scripts

[match-advance-string p409]
[must p419]
[PARSE-ScriptItem p396]

— defun PARSE-Scripts —

```
(defun |PARSE-Scripts| ()
 (and nonblank (match-advance-string "[") (must (|PARSE-ScriptItem|))
      (must (match-advance-string "]"))))
```

———-

8.1.54 defun PARSE-ScriptItem

[PARSE-Expr p383]
[optional p419]
[star p420]
[match-advance-string p409]
[must p419]
[PARSE-ScriptItem p396]
[push-reduction p421]
[pop-stack-2 p495]
[pop-stack-1 p495]

— defun PARSE-ScriptItem —

```
(defun |PARSE-ScriptItem| ()
  (or (and (|PARSE-Expr| 90)
```

```
          (optional
            (and (star repeator
                       (and (match-advance-string ";")
                            (must (|PARSE-ScriptItem|))))
                 (push-reduction '|PARSE-ScriptItem|
                     (cons '|;|
                           (cons (pop-stack-2)
                                 (append (pop-stack-1) nil)))))))
     (and (match-advance-string ";") (must (|PARSE-ScriptItem|))
          (push-reduction '|PARSE-ScriptItem|
            (list '|PrefixSC| (pop-stack-1))))))
```

———-

8.1.55 defun PARSE-Name

[parse-identifier p492]
[push-reduction p421]
[pop-stack-1 p495]

— defun PARSE-Name —

```
(defun |PARSE-Name| ()
  (and (parse-identifier) (push-reduction '|PARSE-Name| (pop-stack-1))))
```

———-

8.1.56 defun PARSE-Data

[action p419]
[PARSE-Sexpr p397]
[push-reduction p421]
[translabel p489]
[pop-stack-1 p495]
[labasoc p**??**]

— defun PARSE-Data —

```
(defun |PARSE-Data| ()
  (declare (special lablasoc))
  (and (action (setq lablasoc nil)) (|PARSE-Sexpr|)
       (push-reduction '|PARSE-Data|
          (list 'quote (translabel (pop-stack-1) lablasoc)))))
```

———-

8.1.57 defun PARSE-Sexpr

[PARSE-Sexpr1 p398]

— **defun PARSE-Sexpr** —

```
(defun |PARSE-Sexpr| ()
  (and (action (advance-token)) (|PARSE-Sexpr1|)))
```

———-

8.1.58 defun PARSE-Sexpr1

[PARSE-AnyId p399]
[optional p419]
[PARSE-NBGliphTok p399]
[must p419]
[PARSE-Sexpr1 p398]
[action p419]
[pop-stack-2 p495]
[nth-stack p496]
[match-advance-string p409]
[push-reduction p421]
[PARSE-IntegerTok p394]
[pop-stack-1 p495]
[PARSE-String p395]
[bang p??]
[star p420]
[PARSE-GliphTok p399]

— **defun PARSE-Sexpr1** —

```
(defun |PARSE-Sexpr1| ()
  (or (and (|PARSE-AnyId|)
           (optional
               (and (|PARSE-NBGliphTok| '=) (must (|PARSE-Sexpr1|))
                    (action (setq lablasoc
                                  (cons (cons (pop-stack-2)
                                              (nth-stack 1))
                                        lablasoc))))))
      (and (match-advance-string "'") (must (|PARSE-Sexpr1|))
           (push-reduction '|PARSE-Sexpr1|
               (list 'quote (pop-stack-1))))
      (|PARSE-IntegerTok|)
      (and (match-advance-string "-") (must (|PARSE-IntegerTok|))
           (push-reduction '|PARSE-Sexpr1| (- (pop-stack-1))))
      (|PARSE-String|)
      (and (match-advance-string "<")
           (bang fil_test (optional (star repeator (|PARSE-Sexpr1|))))
           (must (match-advance-string ">"))
           (push-reduction '|PARSE-Sexpr1| (list2vec (pop-stack-1))))
      (and (match-advance-string "(")
           (bang fil_test
                 (optional
                     (and (star repeator (|PARSE-Sexpr1|))
                          (optional
```

```
                              (and (|PARSE-GliphTok| '|.|)
                                   (must (|PARSE-Sexpr1|))
                                   (push-reduction '|PARSE-Sexpr1|
                                    (nconc (pop-stack-2) (pop-stack-1))))))))
           (must (match-advance-string ")")))))
```

———-

8.1.59 defun PARSE-NBGliphTok

[match-current-token p413]
[action p419]
[advance-token p415]
[tok p375]

— defun PARSE-NBGliphTok —

```
(defun |PARSE-NBGliphTok| (|tok|)
 (declare (special |tok|))
 (and (match-current-token 'gliph |tok|) nonblank (action (advance-token))))
```

———-

8.1.60 defun PARSE-GliphTok

[match-current-token p413]
[action p419]
[advance-token p415]
[tok p375]

— defun PARSE-GliphTok —

```
(defun |PARSE-GliphTok| (|tok|)
 (declare (special |tok|))
 (and (match-current-token 'gliph |tok|) (action (advance-token))))
```

———-

8.1.61 defun PARSE-AnyId

[parse-identifier p492]
[match-string p408]
[push-reduction p421]
[current-symbol p414]
[action p419]
[advance-token p415]
[parse-keyword p493]

— defun PARSE-AnyId —

```
(defun |PARSE-AnyId| ()
  (or (parse-identifier)
      (or (and (match-string "$")
               (push-reduction '|PARSE-AnyId| (current-symbol))
               (action (advance-token)))
          (parse-keyword))))
```

———-

8.1.62 defun PARSE-Sequence

[PARSE-OpenBracket p401]
[must p419]
[PARSE-Sequence1 p400]
[match-advance-string p409]
[PARSE-OpenBrace p401]
[push-reduction p421]
[pop-stack-1 p495]

— defun PARSE-Sequence —

```
(defun |PARSE-Sequence| ()
  (or (and (|PARSE-OpenBracket|) (must (|PARSE-Sequence1|))
           (must (match-advance-string "]")))
      (and (|PARSE-OpenBrace|) (must (|PARSE-Sequence1|))
           (must (match-advance-string "}"))
           (push-reduction '|PARSE-Sequence|
               (list '|brace| (pop-stack-1))))))
```

———-

8.1.63 defun PARSE-Sequence1

[PARSE-Expression p382]
[push-reduction p421]
[pop-stack-2 p495]
[pop-stack-1 p495]
[optional p419]
[PARSE-IteratorTail p402]

— defun PARSE-Sequence1 —

```
(defun |PARSE-Sequence1| ()
  (and (or (and (|PARSE-Expression|)
                (push-reduction '|PARSE-Sequence1|
                    (list (pop-stack-2) (pop-stack-1))))
           (push-reduction '|PARSE-Sequence1| (list (pop-stack-1))))
       (optional
           (and (|PARSE-IteratorTail|)
                (push-reduction '|PARSE-Sequence1|
                    (cons 'collect
```

```
                          (append (pop-stack-1)
                                  (list (pop-stack-1)))))))))
```

———-

8.1.64 defun PARSE-OpenBracket

[getToken p412]
[current-symbol p414]
[eqcar p??]
[push-reduction p421]
[action p419]
[advance-token p415]

— defun PARSE-OpenBracket —

```
(defun |PARSE-OpenBracket| ()
 (let (g1)
  (and (eq (|getToken| (setq g1 (current-symbol))) '[)
       (must (or (and (eqcar g1 '|elt|)
                      (push-reduction '|PARSE-OpenBracket|
                          (list '|elt| (second g1) '|construct|)))
                 (push-reduction '|PARSE-OpenBracket| '|construct|)))
       (action (advance-token)))))
```

———-

8.1.65 defun PARSE-OpenBrace

[getToken p412]
[current-symbol p414]
[eqcar p??]
[push-reduction p421]
[action p419]
[advance-token p415]

— defun PARSE-OpenBrace —

```
(defun |PARSE-OpenBrace| ()
 (let (g1)
  (and (eq (|getToken| (setq g1 (current-symbol))) '{)
       (must (or (and (eqcar g1 '|elt|)
                      (push-reduction '|PARSE-OpenBrace|
                          (list '|elt| (second g1) '|brace|)))
                 (push-reduction '|PARSE-OpenBrace| '|construct|)))
       (action (advance-token)))))
```

———-

8.1.66 defun PARSE-IteratorTail

[match-advance-string p409]
[bang p??]
[optional p419]
[star p420]
[PARSE-Iterator p402]

— defun PARSE-IteratorTail —

```
(defun |PARSE-IteratorTail| ()
 (or (and (match-advance-string "repeat")
          (bang fil_test (optional (star repeator (|PARSE-Iterator|)))))
     (star repeator (|PARSE-Iterator|))))
```

———-

8.1.67 defun PARSE-Iterator

[match-advance-string p409]
[must p419]
[PARSE-Primary p391]
[PARSE-Expression p382]
[PARSE-Expr p383]
[pop-stack-3 p495]
[pop-stack-2 p495]
[pop-stack-1 p495]
[optional p419]

— defun PARSE-Iterator —

```
(defun |PARSE-Iterator| ()
  (or (and (match-advance-string "for") (must (|PARSE-Primary|))
           (must (match-advance-string "in"))
           (must (|PARSE-Expression|))
           (must (or (and (match-advance-string "by")
                          (must (|PARSE-Expr| 200))
                          (push-reduction '|PARSE-Iterator|
                           (list 'inby (pop-stack-3)
                                       (pop-stack-2) (pop-stack-1))))
                     (push-reduction '|PARSE-Iterator|
                         (list 'in (pop-stack-2) (pop-stack-1)))))
           (optional
               (and (match-advance-string "|")
                    (must (|PARSE-Expr| 111))
                    (push-reduction '|PARSE-Iterator|
                        (list '|\|| (pop-stack-1))))))
      (and (match-advance-string "while") (must (|PARSE-Expr| 190))
           (push-reduction '|PARSE-Iterator|
               (list 'while (pop-stack-1))))
      (and (match-advance-string "until") (must (|PARSE-Expr| 190))
           (push-reduction '|PARSE-Iterator|
```

```
            (list 'until (pop-stack-1))))))
```

———-

8.1.68 The PARSE implicit routines

These symbols are not explicitly referenced in the source. Nevertheless, they are called during runtime. For example, PARSE-SemiColon is called in the chain:

```
 PARSE-Enclosure {loc0=nil,loc1="(V  ==> Vector; "} [ihs=35]
  PARSE-Expr
   PARSE-LedPart
    PARSE-Operation
     PARSE-getSemanticForm
      PARSE-SemiColon
```

so there is a bit of indirection involved in the call.

8.1.69 defun PARSE-Suffix

[push-reduction p421]
[current-symbol p414]
[action p419]
[advance-token p415]
[optional p419]
[PARSE-TokTail p387]
[pop-stack-1 p495]

— defun PARSE-Suffix —

```
(defun |PARSE-Suffix| ()
  (and (push-reduction '|PARSE-Suffix| (current-symbol))
       (action (advance-token)) (optional (|PARSE-TokTail|))
       (push-reduction '|PARSE-Suffix|
           (list (pop-stack-1) (pop-stack-1)))))
```

———-

8.1.70 defun PARSE-SemiColon

[match-advance-string p409]
[must p419]
[PARSE-Expr p383]
[push-reduction p421]
[pop-stack-2 p495]
[pop-stack-1 p495]

— defun PARSE-SemiColon —

```
(defun |PARSE-SemiColon| ()
  (and (match-advance-string ";")
```

```
      (must (or (|PARSE-Expr| 82)
                (push-reduction '|PARSE-SemiColon| '|/throwAway|)))
      (push-reduction '|PARSE-SemiColon|
          (list '|;| (pop-stack-2) (pop-stack-1)))))
```

———-

8.1.71 defun PARSE-Return

[match-advance-string p409]
[must p419]
[PARSE-Expression p382]
[push-reduction p421]
[pop-stack-1 p495]

— defun PARSE-Return —

```
(defun |PARSE-Return| ()
  (and (match-advance-string "return") (must (|PARSE-Expression|))
       (push-reduction '|PARSE-Return|
           (list '|return| (pop-stack-1)))))
```

———-

8.1.72 defun PARSE-Exit

[match-advance-string p409]
[must p419]
[PARSE-Expression p382]
[push-reduction p421]
[pop-stack-1 p495]

— defun PARSE-Exit —

```
(defun |PARSE-Exit| ()
  (and (match-advance-string "exit")
       (must (or (|PARSE-Expression|)
                 (push-reduction '|PARSE-Exit| '|$NoValue|)))
       (push-reduction '|PARSE-Exit|
           (list '|exit| (pop-stack-1)))))
```

———-

8.1.73 defun PARSE-Leave

[match-advance-string p409]
[PARSE-Expression p382]
[must p419]
[push-reduction p421]

[PARSE-Label p389]
[pop-stack-1 p495]

— defun PARSE-Leave —

```
(defun |PARSE-Leave| ()
  (and (match-advance-string "leave")
       (must (or (|PARSE-Expression|)
                 (push-reduction '|PARSE-Leave| '|$NoValue|)))
       (must (or (and (match-advance-string "from")
                      (must (|PARSE-Label|))
                      (push-reduction '|PARSE-Leave|
                          (list '|leaveFrom| (pop-stack-1) (pop-stack-1))))
                 (push-reduction '|PARSE-Leave|
                     (list '|leave| (pop-stack-1)))))))
```

———-

8.1.74 defun PARSE-Seg

[PARSE-GliphTok p399]
[bang p??]
[optional p419]
[PARSE-Expression p382]
[push-reduction p421]
[pop-stack-2 p495]
[pop-stack-1 p495]

— defun PARSE-Seg —

```
(defun |PARSE-Seg| ()
  (and (|PARSE-GliphTok| '|..|)
       (bang fil_test (optional (|PARSE-Expression|)))
       (push-reduction '|PARSE-Seg|
           (list 'segment (pop-stack-2) (pop-stack-1)))))
```

———-

8.1.75 defun PARSE-Conditional

[match-advance-string p409]
[must p419]
[PARSE-Expression p382]
[bang p??]
[optional p419]
[PARSE-ElseClause p406]
[push-reduction p421]
[pop-stack-3 p495]
[pop-stack-2 p495]
[pop-stack-1 p495]

— defun PARSE-Conditional —

```
(defun |PARSE-Conditional| ()
  (and (match-advance-string "if") (must (|PARSE-Expression|))
       (must (match-advance-string "then")) (must (|PARSE-Expression|))
       (bang fil_test
             (optional
                 (and (match-advance-string "else")
                      (must (|PARSE-ElseClause|)))))
       (push-reduction '|PARSE-Conditional|
           (list '|if| (pop-stack-3) (pop-stack-2) (pop-stack-1)))))
```

———-

8.1.76 defun PARSE-ElseClause

[current-symbol p414]
[PARSE-Conditional p405]
[PARSE-Expression p382]

— defun PARSE-ElseClause —

```
(defun |PARSE-ElseClause| ()
  (or (and (eq (current-symbol) '|if|) (|PARSE-Conditional|))
      (|PARSE-Expression|)))
```

———-

8.1.77 defun PARSE-Loop

[star p420]
[PARSE-Iterator p402]
[must p419]
[match-advance-string p409]
[PARSE-Expr p383]
[push-reduction p421]
[pop-stack-2 p495]
[pop-stack-1 p495]

— defun PARSE-Loop —

```
(defun |PARSE-Loop| ()
  (or (and (star repeator (|PARSE-Iterator|))
           (must (match-advance-string "repeat"))
           (must (|PARSE-Expr| 110))
           (push-reduction '|PARSE-Loop|
               (cons 'repeat
                     (append (pop-stack-2) (list (pop-stack-1))))))
      (and (match-advance-string "repeat") (must (|PARSE-Expr| 110))
           (push-reduction '|PARSE-Loop|
               (list 'repeat (pop-stack-1))))))
```

———-

8.1.78 defun PARSE-LabelExpr

[PARSE-Label p389]
[must p419]
[PARSE-Expr p383]
[push-reduction p421]
[pop-stack-2 p495]
[pop-stack-1 p495]

— **defun PARSE-LabelExpr** —

```
(defun |PARSE-LabelExpr| ()
  (and (|PARSE-Label|) (must (|PARSE-Expr| 120))
       (push-reduction '|PARSE-LabelExpr|
           (list 'label (pop-stack-2) (pop-stack-1)))))
```

———-

8.1.79 defun PARSE-FloatTok

[parse-number p493]
[push-reduction p421]
[pop-stack-1 p495]
[bfp- p??]
[$boot p??]

— **defun PARSE-FloatTok** —

```
(defun |PARSE-FloatTok| ()
 (declare (special $boot))
  (and (parse-number)
       (push-reduction '|PARSE-FloatTok|
           (if $boot (pop-stack-1) (bfp- (pop-stack-1))))))
```

———-

8.2 The PARSE support routines

This section is broken up into 3 levels:

- String grabbing: Match String, Match Advance String
- Token handling: Current Token, Next Token, Advance Token
- Character handling: Current Char, Next Char, Advance Char
- Line handling: Next Line, Print Next Line
- Error Handling

- Floating Point Support
- Dollar Translation

8.2.1 String grabbing

String grabbing is the art of matching initial segments of the current line, and removing them from the line before the get tokenized if they match (or removing the corresponding current tokens).

8.2.2 defun match-string

The match-string function returns length of X if X matches initial segment of inputstream.
[unget-tokens p412]
[skip-blanks p408]
[match-string line-past-end-p (vol5)]
[match-string current-line (vol5)]
[current-char p416]
[initial-substring-p p410]
[subseq p??]
[$line p??]
[line p??]

— defun match-string —

```
(defun match-string (x)
  (unget-tokens) ; So we don't get out of synch with token stream
  (skip-blanks)
  (if (and (not (line-past-end-p current-line)) (current-char) )
    (initial-substring-p x
     (subseq (line-buffer current-line) (line-current-index current-line)))))
```

———

8.2.3 defun skip-blanks

[current-char p416]
[token-lookahead-type p409]
[skip-blanks advance-char (vol5)]

— defun skip-blanks —

```
(defun skip-blanks ()
  (loop (let ((cc (current-char)))
          (if (not cc) (return nil))
          (if (eq (token-lookahead-type cc) 'white)
              (if (not (advance-char)) (return nil))
              (return t)))))
```

———

— initvars —

```
(defvar Escape-Character #\\ "Superquoting character.")
```

———-

8.2.4 defun token-lookahead-type

[Escape-Character p??]

— defun token-lookahead-type —

```
(defun token-lookahead-type (char)
  "Predicts the kind of token to follow, based on the given initial character."
 (declare (special Escape-Character))
  (cond
   ((not char)                                          'eof)
   ((or (char= char Escape-Character) (alpha-char-p char)) 'id)
   ((digitp char)                                       'num)
   ((char= char #\')                                    'string)
   ((char= char #\[)                                    'bstring)
   ((member char '(#\Space #\Tab #\Return) :test #'char=)  'white)
   (t                                                    'special-char)))
```

———-

8.2.5 defun match-advance-string

The match-string function returns length of X if X matches initial segment of inputstream. If it is successful, advance inputstream past X. [quote-if-string p410]
[current-token p414]
[match-string p408]
[line-current-index p??]
[match-advance-string line-past-end-p (vol5)]
[match-advance-string current-line (vol5)]
[line-current-char p??]
[$token p95]
[$line p??]

— defun match-advance-string —

```
(defun match-advance-string (x)
  (let ((y (if (>= (length (string x))
                   (length (string (quote-if-string (current-token)))))
               (match-string x)
               nil))) ; must match at least the current token
    (when y
      (incf (line-current-index current-line) y)
      (if (not (line-past-end-p current-line))
       (setf (line-current-char current-line)
             (elt (line-buffer current-line)
```

```
                (line-current-index current-line)))
       (setf (line-current-char current-line) #\space))
      (setq prior-token
       (make-token :symbol (intern (string x))
                   :type 'identifier
                   :nonblank nonblank))
      t)))
```

———-

8.2.6 defun initial-substring-p

[string-not-greaterp p??]

— defun initial-substring-p —

```
(defun initial-substring-p (part whole)
  "Returns length of part if part matches initial segment of whole."
  (let ((x (string-not-greaterp part whole)))
    (and x (= x (length part)) x)))
```

———-

8.2.7 defun quote-if-string

[token-type p??]
[strconc p??]
[token-symbol p??]
[underscore p411]
[token-nonblank p??]
[pack p??]
[escape-keywords p411]
[$boot p??]
[$spad p515]

— defun quote-if-string —

```
(defun quote-if-string (token)
 (declare (special $boot $spad))
 (when token   ;only use token-type on non-null tokens
  (case (token-type token)
   (bstring      (strconc "[" (token-symbol token) "]*"))
   (string       (strconc "'" (token-symbol token) "'"))
   (spadstring   (strconc "\"" (underscore (token-symbol token)) "\""))
   (number       (format nil "~v,'0D" (token-nonblank token)
                               (token-symbol token)))
   (special-char (string (token-symbol token)))
   (identifier   (let ((id (symbol-name (token-symbol token)))
                          (pack (package-name (symbol-package
                                                (token-symbol token)))))
                 (if (or $boot $spad)
```

```
                   (if (string= pack "BOOT")
                    (escape-keywords (underscore id) (token-symbol token))
                    (concatenate 'string
                      (underscore pack) "'" (underscore id)))
                   id)))
   (t            (token-symbol token)))))
```

———-

8.2.8 defun escape-keywords

[$keywords p??]

— defun escape-keywords —

```
(defun escape-keywords (pname id)
 (declare (special keywords))
  (if (member id keywords)
   (concatenate 'string "_" pname)
   pname))
```

———-

8.2.9 defun isTokenDelimiter

NIL needed below since END_UNIT is not generated by current parser [current-symbol p414]

— defun isTokenDelimiter —

```
(defun |isTokenDelimiter| ()
 (member (current-symbol) '(\) end\_unit nil)))
```

———-

8.2.10 defun underscore

[vector-push p??]

— defun underscore —

```
(defun underscore (string)
 (if (every #'alpha-char-p string)
  string
  (let* ((size (length string))
         (out-string (make-array (* 2 size)
                                 :element-type 'string-char
                                 :fill-pointer 0))
         next-char)
   (dotimes (i size)
    (setq next-char (char string i))
    (unless (alpha-char-p next-char) (vector-push #\_ out-string))
```

```
    (vector-push next-char out-string))
   out-string)))
```

———-

8.2.11 Token Handling

8.2.12 defun getToken

[eqcar p??]

— defun getToken —

```
(defun |getToken| (x)
 (if (eqcar x '|elt|) (third x) x))
```

———-

8.2.13 defun unget-tokens

[quote-if-string p410]
[unget-tokens line-current-segment (vol5)]
[unget-tokens current-line (vol5)]
[strconc p??]
[line-number p??]
[token-nonblank p??]
[unget-tokens line-new-line (vol5)]
[line-number p??]
[valid-tokens p96]

— defun unget-tokens —

```
(defun unget-tokens ()
 (case valid-tokens
  (0 t)
  (1 (let* ((cursym (quote-if-string current-token))
            (curline (line-current-segment current-line))
            (revised-line (strconc cursym curline (copy-seq " "))))
         (line-new-line revised-line current-line (line-number current-line))
         (setq nonblank (token-nonblank current-token))
         (setq valid-tokens 0)))
  (2 (let* ((cursym (quote-if-string current-token))
            (nextsym (quote-if-string next-token))
            (curline (line-current-segment Current-Line))
            (revised-line
             (strconc (if (token-nonblank current-token) "" " ")
                      cursym
                      (if (token-nonblank next-token) "" " ")
                      nextsym curline " ")))
      (setq nonblank (token-nonblank current-token))
      (line-new-line revised-line current-line (line-number current-line))
```

```
    (setq valid-tokens 0)))
  (t (error "How many tokens do you think you have?"))))
```

———-

8.2.14 defun match-current-token

This returns the current token if it has EQ type and (optionally) equal symbol. [current-token p414]
[match-token p413]

— defun match-current-token —

```
(defun match-current-token (type &optional (symbol nil))
  (match-token (current-token) type symbol))
```

———-

8.2.15 defun match-token

[token-type p??]
[token-symbol p??]

— defun match-token —

```
(defun match-token (token type &optional (symbol nil))
  (when (and token (eq (token-type token) type))
   (if symbol
    (when (equal symbol (token-symbol token)) token)
    token)))
```

———-

8.2.16 defun match-next-token

This returns the next token if it has equal type and (optionally) equal symbol. [next-token p415]
[match-token p413]

— defun match-next-token —

```
(defun match-next-token (type &optional (symbol nil))
  (match-token (next-token) type symbol))
```

———-

8.2.17 defun current-symbol

[make-symbol-of p414]
[current-token p414]

— defun current-symbol —

```
(defun current-symbol ()
 (make-symbol-of (current-token)))
```

——–

8.2.18 defun make-symbol-of

[$token p95]

— defun make-symbol-of —

```
(defun make-symbol-of (token)
 (let ((u (and token (token-symbol token))))
  (cond
   ((not u) nil)
   ((characterp u) (intern (string u)))
   (u))))
```

——–

8.2.19 defun current-token

This returns the current token getting a new one if necessary. [try-get-token p414]
[valid-tokens p96]
[current-token p414]

— defun current-token —

```
(defun current-token ()
 (declare (special valid-tokens current-token))
 (if (> valid-tokens 0)
  current-token
  (try-get-token current-token)))
```

——–

8.2.20 defun try-get-token

[get-token p416]
[valid-tokens p96]

— defun try-get-token —

```
(defun try-get-token (token)
```

```
 (declare (special valid-tokens))
 (let ((tok (get-token token)))
  (when tok
   (incf valid-tokens)
   token)))
```

———-

8.2.21 defun next-token

This returns the token after the current token, or NIL if there is none after. [try-get-token p414]
[current-token p414]
[valid-tokens p96]
[next-token p415]

— defun next-token —

```
(defun next-token ()
 (declare (special valid-tokens next-token))
 (current-token)
 (if (> valid-tokens 1)
  next-token
  (try-get-token next-token)))
```

———-

8.2.22 defun advance-token

This makes the next token be the current token. [current-token p414]
[copy-token p??]
[try-get-token p414]
[valid-tokens p96]
[current-token p414]

— defun advance-token —

```
(defun advance-token ()
  (current-token)                        ;don't know why this is needed
  (case valid-tokens
    (0 (try-get-token (current-token)))
    (1 (decf valid-tokens)
       (setq prior-token (copy-token current-token))
       (try-get-token current-token))
    (2 (setq prior-token (copy-token current-token))
       (setq current-token (copy-token next-token))
       (decf valid-tokens))))
```

———-

8.2.23 defvar XTokenReader

— initvars —

```
(defvar XTokenReader 'get-meta-token "Name of tokenizing function")
```

———-

8.2.24 defun get-token

[XTokenReader p416]
[XTokenReader p416]

— defun get-token —

```
(defun get-token (token)
 (funcall XTokenReader token))
```

———-

8.2.25 Character handling

8.2.26 defun current-char

This returns the current character of the line, initially blank for an unread line. [$line p??]
[current-char line-past-end-p (vol5)]
[current-char current-line (vol5)]
[current-line p??]

— defun current-char —

```
(defun current-char ()
  (if (line-past-end-p current-line)
   #\return
   (line-current-char current-line)))
```

———-

8.2.27 defun next-char

This returns the character after the current character, blank if at end of line. The blank-at-end-of-line assumption is allowable because we assume that end-of-line is a token separator, which blank is equivalent to. [next-char line-at-end-p (vol5)]
[next-char line-next-char (vol5)]
[next-char current-line (vol5)]

— defun next-char —

```
(defun next-char ()
```

```
 (if (line-at-end-p current-line)
  #\return
  (line-next-char current-line)))
```

———-

8.2.28 defun char-eq

— defun char-eq —

```
(defun char-eq (x y)
 (char= (character x) (character y)))
```

———-

8.2.29 defun char-ne

— defun char-ne —

```
(defun char-ne (x y)
 (char/= (character x) (character y)))
```

———-

8.2.30 Error handling

8.2.31 defvar meta-error-handler

— initvars —

```
(defvar meta-error-handler 'meta-meta-error-handler)
```

———-

8.2.32 defun meta-syntax-error

[meta-error-handler p417]
[meta-error-handler p417]

— defun meta-syntax-error —

```
(defun meta-syntax-error (&optional (wanted nil) (parsing nil))
  (declare (special meta-error-handler))
  (funcall meta-error-handler wanted parsing))
```

———-

8.2.33 Floating Point Support

8.2.34 defun floatexpid

TPDHERE: The use of and in spadreduce is undefined. rewrite this to loop
[floatexpid identp (vol5)]
[floatexpid pname (vol5)]
[spadreduce p??]
[collect p349]
[step p??]
[maxindex p??]
[floatexpid digitp (vol5)]

— defun floatexpid —

```
(defun floatexpid (x &aux s)
 (when (and (identp x) (char= (char-upcase (elt (setq s (pname x)) 0)) #\E)
            (> (length s) 1)
            (spadreduce and 0 (collect (step i 1 1 (maxindex s))
                                       (digitp (elt s i)))))
   (read-from-string s t nil :start 1)))
```

———

8.2.35 Dollar Translation

8.2.36 defun dollarTran

[$InteractiveMode p??]

— defun dollarTran —

```
(defun |dollarTran| (dom rand)
 (let ((eltWord (if |$InteractiveMode| '|$elt| '|elt|)))
  (declare (special |$InteractiveMode|))
  (if (and (not (atom rand)) (cdr rand))
   (cons (list eltWord dom (car rand)) (cdr rand))
   (list eltWord dom rand))))
```

———

8.2.37 Applying metagrammatical elements of a production (e.g., Star).

- **must** means that if it is not present in the token stream, it is a syntax error.
- **optional** means that if it is present in the token stream, that is a good thing, otherwise don't worry (like [foo] in BNF notation).
- **action** is something we do as a consequence of successful parsing; it is inserted at the end of the conjunction of requirements for a successful parse, and so should return T.

- **sequence** consists of a head, which if recognized implies that the tail must follow. Following tail are actions, which are performed upon recognizing the head and tail.

8.2.38 defmacro Bang

If the execution of prod does not result in an increase in the size of the stack, then stack a NIL. Return the value of prod.

— defmacro bang —

```
(defmacro bang (lab prod)
 `(progn
   (setf (stack-updated reduce-stack) nil)
   (let* ((prodvalue ,prod) (updated (stack-updated reduce-stack)))
    (unless updated (push-reduction ',lab nil))
    prodvalue)))
```

———-

8.2.39 defmacro must

[meta-syntax-error p417]

— defmacro must —

```
(defmacro must (dothis &optional (this-is nil) (in-rule nil))
  `(or ,dothis (meta-syntax-error ,this-is ,in-rule)))
```

———-

8.2.40 defun action

— defun action —

```
(defun action (dothis) (or dothis t))
```

———-

8.2.41 defun optional

— defun optional —

```
(defun optional (dothis) (or dothis t))
```

———-

8.2.42 defmacro star

Succeeds if there are one or more of PROD, stacking as one unit the sub-reductions of PROD and labelling them with LAB. E.G., `(Star IDs (parse-id))` with A B C will stack (3 IDs (A B C)), where (parse-id) would stack (1 ID (A)) when applied once. [stack-size p??]
[push-reduction p421]
[pop-stack-1 p495]

— **defmacro star** —

```
(defmacro star (lab prod)
  `(prog ((oldstacksize (stack-size reduce-stack)))
     (if (not ,prod) (return nil))
loop
     (if (not ,prod)
      (let* ((newstacksize (stack-size reduce-stack))
             (number-of-new-reductions (- newstacksize oldstacksize)))
        (if (> number-of-new-reductions 0)
         (return (do ((i 0 (1+ i)) (accum nil))
                     ((= i number-of-new-reductions)
                       (push-reduction ',lab accum)
                       (return t))
                   (push (pop-stack-1) accum)))
         (return t)))
      (go loop))))
```

———-

8.2.43 Stacking and retrieving reductions of rules.

8.2.44 defvar reduce-stack

Stack of results of reduced productions. [$stack p93]

— **initvars** —

```
(defvar reduce-stack (make-stack) )
```

———-

8.2.45 defmacro reduce-stack-clear

— **defmacro reduce-stack-clear** —

```
(defmacro reduce-stack-clear () `(stack-load nil reduce-stack))
```

———-

8.2.46 defun push-reduction

[stack-push p94]
[make-reduction p??]
[reduce-stack p420]

— defun push-reduction —

```
(defun push-reduction (rule redn)
  (stack-push (make-reduction :rule rule :value redn) reduce-stack))
```

———-

Chapter 9

Comment Recording

This is the graph of the functions used for recording comments. The syntax is a graphviz dot file. To generate this graph as a JPEG file, type:

```
tangle v9CommentRecording.dot bookvol9.pamphlet >v9cr.dot
dot -Tjpg v9cr.dot >v9cr.jpg
```

— **v9CommentRecording.dot** —

```
digraph pic {
 fontsize=10;
 bgcolor="#ECEA81";
 node [shape=box, color=white, style=filled];

"postDef"        [color="#FFFFFF"]
"PARSE-Category"        [color="#FFFFFF"]

"recordAttributeDocumentation"  [color="#FF6600"]
"recordSignatureDocumentation"  [color="#FF6600"]
"recordDocumentation"  [color="#2222DD"]
"collectComBlock"  [color="#22EE22"]
"recordHeaderDocumentation"  [color="#FFFF66"]
"collectAndDeleteAssoc"  [color="#FFFF66"]

"postDef"                        ->  "recordHeaderDocumentation"
"PARSE-Category"                 ->  "recordSignatureDocumentation"
"PARSE-Category"                 ->  "recordAttributeDocumentation"

"recordAttributeDocumentation"  ->  "recordDocumentation"
"recordSignatureDocumentation"  ->  "recordDocumentation"
"recordDocumentation"  ->  "recordHeaderDocumentation"
"recordDocumentation"  ->  "collectComBlock"
"collectComBlock"  ->  "collectAndDeleteAssoc"
}
```

———

9.1 Comment Recording Layer 0 – API

9.1.1 defun recordSignatureDocumentation

This function is called externally by PARSE-Category. [recordDocumentation p424]
[postTransform p337]

— defun recordSignatureDocumentation —

```
(defun |recordSignatureDocumentation| (opSig lineno)
  (|recordDocumentation| (cdr (postTransform opSig)) lineno))
```

———-

9.1.2 defun recordAttributeDocumentation

This function is called externally by PARSE-Category. [opOf p??]
[pname p??]
[upper-case-p p??]
[recordDocumentation p424]
[ifcdr p??]
[postTransform p337]

— defun recordAttributeDocumentation —

```
(defun |recordAttributeDocumentation| (arg lineno)
 (let (att name)
  (setq att (cadr arg))
  (setq name (|opOf| att))
  (cond
   ((upper-case-p (elt (pname name) 0)) nil)
   (t
    (|recordDocumentation|
     (list name (cons '|attribute| (ifcdr (postTransform att)))) lineno)))))
```

———-

9.2 Comment Recording Layer 1

9.2.1 defun recordDocumentation

[recordHeaderDocumentation p425]
[collectComBlock p425]
[$maxSignatureLineNumber p??]
[$docList p??]

— defun recordDocumentation —

```
(defun |recordDocumentation| (key lineno)
 (let (u)
```

```
 (declare (special |$docList| |$maxSignatureLineNumber|))
  (|recordHeaderDocumentation| lineno)
  (setq u (|collectComBlock| lineno))
  (setq |$maxSignatureLineNumber| lineno)
  (setq |$docList| (cons (cons key u) |$docList|))))
```

———-

9.3 Comment Recording Layer 2

9.3.1 defun collectComBlock

[collectAndDeleteAssoc p426]
[$comblocklist p498]

— defun collectComBlock —

```
(defun |collectComBlock| (x)
 (let (val u)
 (declare (special $comblocklist))
  (cond
   ((and (consp $comblocklist)
         (consp (qcar $comblocklist))
         (equal (qcaar $comblocklist) x))
     (setq val (qcdar $comblocklist))
     (setq u (append val (|collectAndDeleteAssoc| x)))
     (setq $comblocklist (cdr $comblocklist))
     u)
   (t (|collectAndDeleteAssoc| x)))))
```

———-

9.4 Comment Recording Layer 3

9.4.1 defun recordHeaderDocumentation

This function is called externally by postDef. [assocright p??]
[$maxSignatureLineNumber p??]
[$comblocklist p498]
[$headerDocumentation p??]
[$headerDocumentation p??]
[$comblocklist p498]

— defun recordHeaderDocumentation —

```
(defun |recordHeaderDocumentation| (lineno)
 (let (al)
 (declare (special |$headerDocumentation| |$maxSignatureLineNumber|
                   $comblocklist))
```

```
  (when (eql |$maxSignatureLineNumber| 0)
    (setq al
      (loop for p in $comblocklist
       when (or (null (car p)) (null lineno) (> lineno (car p)))
       collect p))
    (setq $comblocklist (setdifference $comblocklist al))
    (setq |$headerDocumentation| (assocright al))
    (when |$headerDocumentation| (setq |$maxSignatureLineNumber| 1))
    |$headerDocumentation|)))
```

———-

9.4.2 defun collectAndDeleteAssoc

```
   u is (.. (x . a) .. (x . b) .. )  ==> (a b ..)
```

deleting entries from u assumes that the first element is useless [$comblocklist p498]

— defun collectAndDeleteAssoc —

```
(defun |collectAndDeleteAssoc| (x)
 (let (r res s)
 (declare (special $comblocklist))
  (maplist
   #'(lambda (y)
      (when (setq s (cdr y))
       (do ()
           ((null (and s (consp (car s)) (equal (qcar (car s)) x))) nil)
        (setq r (qcdr (car s)))
        (setq res (append res r))
        (setq s (cdr s))
        (rplacd y s))))
    $comblocklist)
  res))
```

———-

Chapter 10

Category handling

10.0.3 defun getConstructorExports

— defun getConstructorExports —

```
(defun |getConstructorExports| (&rest arg)
 (let (options conform)
  (setq conform (car arg))
  (setq options (cdr arg))
  (|categoryParts| conform
   (getdatabase (|opOf| conform) 'constructorcategory)
    (ifcar options))))
```

———

Chapter 11

Building libdb.text

11.0.4 defun extendLocalLibdb

[buildLibdb p430]
[union p??]
[purgeNewConstructorLines p432]
[dbReadLines p432]
[dbWriteLines p433]
[extendLocalLibdb deleteFile (vol5)]
[msort p??]
[$createLocalLibDb p??]
[$newConstructorList p??]
[$newConstructorList p??]

— defun extendLocalLibdb —

```
(defun |extendLocalLibdb| (conlist)
 (let (localLibdb oldlines newlines)
 (declare (special |$createLocalLibDb| |$newConstructorList|))
  (cond
   ((null |$createLocalLibDb|) nil)
   ((null conlist) nil)
   (t
     (|buildLibdb| conlist)
     (setq |$newConstructorList| (|union| conlist |$newConstructorList|))
     (setq localLibdb "libdb.text")
     (cond
      ((null (probe-file "libdb.text"))
        (rename-file "temp.text" "libdb.text"))
      (t
       (setq oldlines
         (|purgeNewConstructorLines| (|dbReadLines| localLibdb) conlist))
       (setq newlines (|dbReadLines| "temp.text"))
       (|dbWriteLines| (msort (|union| oldlines newlines)) "libdb.text")
       (|deleteFile| "temp.text")))))))
```

11.0.5 defun buildLibdb

This function appears to have two use cases, one in which the domainList variable is undefined, in which case it writes out all of the constructors, and the other case where it writes out a single constructor. Formal for libdb.text:

```
constructors    Cname\#\I\sig \args   \abb \comments (C is C, D, P, X)
operations      Op  \#\E\sig \conname\pred\comments (E is one of U/E)
attributes      Aname\#\E\args\conname\pred\comments
I = <x if exposed><d if category with a default package>
```

[dsetq p??]
[ifcar p??]
[buildLibdb deleteFile (vol5)]
[buildLibdb make-outstream (vol5)]
[writedb p??]
[buildLibdbString p432]
[buildLibdb allConstructors (vol5)]
[buildLibdbConEntry p433]
[getConstructorExports p427]
[buildLibOps p435]
[buildLibAttrs p436]
[shut p??]
[obey p??]
[deleteFile p??]
[$outStream p??]
[$conform p??]
[$kind p??]
[$doc p??]
[$exposed? p??]
[$conform p??]
[$conname p??]
[$outStream p??]
[$DefLst p??]
[$PakLst p??]
[$catLst p??]
[$DomLst p??]
[$AttrLst p??]
[$OpLst p??]

— defun buildLibdb —

```
(defun |buildLibdb| (&rest G168131 &AUX options)
 (dsetq options G168131)
 (let (|$OpLst| |$AttrLst| |$DomLst| |$CatLst| |$PakLst| |$DefLst|
       |$outStream| |$conname| |$conform| |$exposed?| |$doc|
       |$kind| domainList comments constructorList tmp1 attrlist oplist)
 (declare (special |$OpLst| |$AttrLst| |$DomLst| |$CatLst| |$PakLst|
                   |$DefLst| |$outStream| |$conname| |$conform|
                   |$exposed?| |$doc| |$kind|))
  (setq domainList (ifcar options))
  (setq |$OpLst| nil)
  (setq |$AttrLst| nil)
```

```
(setq |$DomLst| nil)
(setq |$CatLst| nil)
(setq |$PakLst| nil)
(setq |$DefLst| nil)
(|deleteFile| "temp.text")
(setq |$outStream| (make-outstream "temp.text"))
(unless domainList
 (setq comments
  (concatenate 'string
   "\\spad{Union(A,B,...,C)} is a primitive type in AXIOM used to "
   "represent objects of type \\spad{A} or of type \\spad{B} or...or "
   "of type \\spad{C}."))
 (|writedb|
  (|buildLibdbString|
   (list "dUnion" 1 "x" "special" "(A,B,...,C)" 'UNION comments)))
 (setq comments
  (concatenate 'string
    "\\spad{Record(a:A,b:B,...,c:C)} is a primitive type in AXIOM used "
    "to represent composite objects made up of objects of type "
    "\\spad{A}, \\spad{B},..., \\spad{C} which are indexed by \"keys\""
    " (identifiers) \\spad{a},\\spad{b},...,\\spad{c}."))
 (|writedb|
  (|buildLibdbString|
   (list "dRecord" 1 "x" "special" "(a:A,b:B,...,c:C)" 'RECORD comments)))
 (setq comments
  (concatenate 'string
   "\\spad{Mapping(T,S)} is a primitive type in AXIOM used to represent"
   " mappings from source type \\spad{S} to target type \\spad{T}. "
   "Similarly, \\spad{Mapping(T,A,B)} denotes a mapping from source "
   "type \\spad{(A,B)} to target type \\spad{T}."))
 (|writedb|
  (|buildLibdbString|
   (list "dMapping" 1 "x" "special" "(T,S)" 'MAPPING comments)))
 (setq comments
  (concatenate 'string
   "\\spad{Enumeration(a,b,...,c)} is a primitive type in AXIOM used to "
   "represent the object composed of the symbols \\spad{a},\\spad{b},"
   "..., and \\spad{c}."))
 (|writedb|
  (|buildLibdbString|
   (list "dEnumeration" 1 "x" "special" "(a,b,...,c)" 'ENUM comments))))
(setq |$conname| nil)
(setq |$conform| nil)
(setq |$exposed?| nil)
(setq |$doc| nil)
(setq |$kind| nil)
(setq constructorList (or domainList (|allConstructors|)))
(loop for con in constructorList do
  (|writedb| (|buildLibdbConEntry| con))
  (setq tmp1 (|getConstructorExports| |$conform|))
  (setq attrlist (car tmp1))
  (setq oplist (cdr tmp1))
  (|buildLibOps| oplist)
  (|buildLibAttrs| attrlist))
```

```
   (shut |$outStream|)
   (unless domainList
     (obey "sort  \"temp.text\"  > \"libdb.text\"")
     (rename-file "libdb.text" "olibdb.text")
     (|deleteFile| "temp.text"))))
```

———-

11.0.6 defun buildLibdbString

[strconc p??]

— defun buildLibdbString —

```
(defun |buildLibdbString| (arg)
 (let (x u)
  (setq x (car arg))
  (setq u (cdr arg))
  (strconc (princ-to-string x)
   (let ((result ""))
    (loop for y in u
     collect (setq result (strconc result (strconc "‘" (princ-to-string y)))))
    result))))
```

———-

11.0.7 defun dbReadLines

[eofp p??]
[readline p??]

— defun dbReadLines —

```
(defun |dbReadLines| (target)
 (let (instream lines)
  (setq instream (open target))
  (setq lines
   (loop while (not (eofp instream))
    collect (readline instream)))
   (close instream)
   lines))
```

———-

11.0.8 defun purgeNewConstructorLines

[screenLocalLine p437]

— defun purgeNewConstructorLines —

```
(defun |purgeNewConstructorLines| (lines conlist)
```

```
(loop for x in lines
 when (null (|screenLocalLine| x conlist))
 collect x))
```

———-

11.0.9 defun dbWriteLines

[ifcar p??]
[getTempPath p??]
[make-outstream p??]
[writedb p??]
[shut p??]
[$outStream p??]
[$outStream p??]

— defun dbWriteLines —

```
(defun |dbWriteLines| (&rest G176369 &aux options s)
 (dsetq (s . options) G176369)
 (let (|$outStream| pathname)
 (declare (special |$outStream|))
  (setq pathname (or (ifcar options) (|getTempPath| '|source|)))
  (setq |$outStream| (make-outstream pathname))
  (loop for x in s do (|writedb| x))
  (shut |$outStream|)
  pathname))
```

———-

11.0.10 defun buildLibdbConEntry

[getdatabase p??]
[dbMkForm p??]
[msubst p??]
[isExposedConstructor p??]
[pname p??]
[maxindex p??]
[downcase p??]
[lassoc p??]
[libdbTrim p??]
[concatWithBlanks p??]
[form2HtString p??]
[libConstructorSig p??]
[strconc p??]
[buildLibdbString p432]
[length p??]
[$exposed? p??]
[$kind p??]

[$conform p??]
[$kind p??]
[$doc p??]
[$exposed? p??]
[$conname p??]

— defun buildLibdbConEntry —

```
(defun |buildLibdbConEntry| (conname)
 (let (abb conform pname kind argl tmp1 conComments argpart sigpart header)
 (declare (special |$exposed?| |$doc| |$kind| |$conname| |$conform|))
  (cond
   ((null (getdatabase conname 'constructormodemap)) nil)
   (t
    (setq abb (getdatabase conname 'abbreviation))
    (setq |$conname| conname)
    (setq conform (or (getdatabase conname 'constructorform) (list conname)))
    (setq |$conform| (|dbMkForm| (msubst 't 'T$ conform)))
    (cond
     ((null |$conform|) nil)
     (t
      (setq |$exposed?| (if (|isExposedConstructor| conname) "x" "n"))
      (setq |$doc| (getdatabase conname 'documentation))
      (setq pname (pname conname))
      (setq kind (getdatabase conname 'constructorkind))
      (cond
       ((and (eq kind '|domain|)
             (progn
              (setq tmp1 (getdatabase conname 'constructormodemap))
              (and (consp tmp1)
                   (consp (qcar tmp1))
                   (consp (qcdar tmp1))))
             (consp (qcadar tmp1)) (eq (qcaadar tmp1) 'category)
             (progn
              (and (consp (qcdadar tmp1))
                   (eq (qcar (qcdadar tmp1)) '|package|))))
         (setq kind '|package|)))
     (setq |$kind|
       (if (char= (elt pname (maxindex pname)) #\&)
        '|x|
        (downcase (elt (pname kind) 0))))
     (setq argl (cdr |$conform|))
     (setq conComments
      (cond
       ((progn
         (setq tmp1 (lassoc '|constructor| |$doc|))
         (and (consp tmp1)
              (eq (qcdr tmp1) nil)
              (consp (qcar tmp1))
              (equal (qcaar tmp1) nil)))
         (|libdbTrim| (|concatWithBlanks| (qcdar tmp1))))
       (t "")))
     (setq argpart (substring (|form2HtString| (cons '|f| argl)) 1 nil))
     (setq sigpart (|libConstructorSig| |$conform|))
```

```
     (setq header (strconc |$kind| (pname conname)))
     (|buildLibdbString|
       (list header (|#| argl) |$exposed?|
             sigpart argpart abb conComments))))))))
```

———-

11.0.11 defun buildLibOps

[buildLibOp p435]

— defun buildLibOps —

```
(defun |buildLibOps| (oplist)
 (loop for item in oplist
   do (|buildLibOp| (car item) (cadr item) (cddr item))))
```

———-

11.0.12 defun buildLibOp

[sublislis p??]
[msubst p??]
[form2LispString p??]
[strconc p??]
[libdbTrim p??]
[concatWithBlanks p??]
[lassoc p??]
[checkCommentsForBraces p??]
[writedb p??]
[buildLibdbString p432]
[$kind p??]
[$doc p??]
[$exposed? p??]
[$conform p??]

— defun buildLibOp —

```
(defun |buildLibOp| (op sig pred)
 (let (nsig sigpart predString s sop header conform comments)
 (declare (special |$kind| |$doc| |$exposed?| |$conform|))
  (setq nsig (sublislis (cdr |$conform|) |$FormalMapVariableList| sig))
  (setq pred (sublislis (cdr |$conform|) |$FormalMapVariableList| pred))
  (setq nsig (msubst 't 't$ nsig))
  (setq pred (msubst 't 't$ pred))
  (setq sigpart (|form2LispString| (cons '|Mapping| nsig)))
  (setq predString (if (eq pred t) "" (|form2LispString| pred)))
  (setq sop
   (cond
    ((string= (setq s (princ-to-string op)) "One") "1")
    ((string= s "Zero") "0")
```

```
    (t s)))
  (setq header (strconc "o" sop))
  (setq conform (strconc |$kind| (|form2LispString| |$conform|)))
  (setq comments
    (|libdbTrim| (|concatWithBlanks| (lassoc sig (lassoc op |$doc|)))))
  (|checkCommentsForBraces| '|operation| sop sigpart comments)
  (|writedb|
   (|buildLibdbString|
    (list header (|#| (cdr sig)) |$exposed?| sigpart
          conform predString comments)))))
```

———-

11.0.13 defun buildLibAttrs

[buildLibAttr p436]

— defun buildLibAttrs —

```
(defun |buildLibAttrs| (attrlist)
 (let (name argl pred)
  (loop for item in attrlist
   do (|buildLibAttr| (car item) (cadr item) (cddr item)))))
```

———-

11.0.14 defun buildLibAttr

```
  attributes      AKname\#\args\conname\pred\comments (K is U or C)
```

[form2LispString p??]
[sublislis p??]
[concatWithBlanks p??]
[lassoc p??]
[checkCommentsForBraces p??]
[writedb p??]
[buildLibdbString p432]
[length p??]
[$conform p??]
[$FormalMapVariableList p249]
[$kind p??]
[$doc p??]
[$exposed? p??]
[$conname p??]

— defun buildLibAttr —

```
(defun |buildLibAttr| (name argl pred)
 (let (argPart predString header conname comments)
 (declare (special |$kind| |$conname| |$doc| |$conform|
                   |$FormalMapVariableList| |$exposed?|))
```

```
  (setq header (strconc "a" (princ-to-string name)))
  (setq argPart (substring (|form2LispString| (cons '|f| argl)) 1 nil))
  (setq pred (sublislis (cdr |$conform|) |$FormalMapVariableList| pred))
  (setq predString (if (eq pred t) "" (|form2LispString| pred)))
  (setq header (strconc "a" (princ-to-string name)))
  (setq conname (strconc |$kind| (|form2LispString| |$conname|)))
  (setq comments
   (|concatWithBlanks| (lassoc (cons '|attribute| argl) (lassoc name |$doc|))))
  (|checkCommentsForBraces| '|attribute| (princ-to-string name) argl comments)
  (|writedb|
   (|buildLibdbString|
    (list header (|#| argl) |$exposed?| argPart
          conname predString comments)))))
```

———

11.0.15 defun screenLocalLine

[dbPart p??]
[charPosition p??]
[dbName p??]
[dbKind p??]

— defun screenLocalLine —

```
(defun |screenLocalLine| (line conlist)
 (let (s k con)
  (setq k (|dbKind| line))
  (setq con
   (intern (cond ((or (char= k #\o) (char= k #\a))
                  (setq s (|dbPart| line 5 1))
                  (setq k (|charPosition| #\( s 1))
                  (substring s 1 (1- k)))
                (t (|dbName| line)))))
  (member con conlist)))
```

———

Chapter 12

Comment Syntax Checking

This is the graph of the functions used for comment syntax checking. The syntax is a graphviz dot file. To generate this graph as a JPEG file, type:

```
tangle v9CommentSyntaxChecking.dot bookvol9.pamphlet >v9csc.dot
dot -Tjpg v9csc.dot >v9csc.jpg
```

— **v9CommentSyntaxChecking.dot** —

```
digraph hierarchy {
 fontsize=10;
 bgcolor="#ECEA81";
 node [shape=box, color=white, style=filled];

"compileDocumentation" [color="#FFFFFF"]
"finalizeLisplib"      [color="#FFFFFF"]

{rank=same; "compileDocumentation" "finalizeLisplib"}

"checkAddBackSlashes"  [color="#FFFF66"]
"checkAddMacros"  [color="#FFFF66"]
"checkAddPeriod"  [color="#FFFF66"]
"checkAddSpaceSegments"  [color="#FFFF66"]
"checkAddSpaces"  [color="#FFFF66"]
"checkAlphabetic"  [color="#FFFF66"]
"checkIeEgfun"  [color="#FFFF66"]
"checkIsValidType"  [color="#FFFF66"]
"checkLookForLeftBrace"  [color="#FFFF66"]
"checkLookForRightBrace"  [color="#FFFF66"]
"checkNumOfArgs"  [color="#FFFF66"]
"checkSayBracket"  [color="#FFFF66"]
"checkSkipBlanks"  [color="#FFFF66"]
"checkSplitBackslash"  [color="#FFFF66"]
"checkSplitOn"  [color="#FFFF66"]
"checkSplitPunctuation"  [color="#FFFF66"]
"firstNonBlankPosition"  [color="#FFFF66"]
"getMatchingRightPren"  [color="#FFFF66"]
"hasNoVowels"  [color="#FFFF66"]
```

```
"htcharPosition"  [color="#FFFF66"]
"newWordFrom"  [color="#FFFF66"]
"removeBackslashes"  [color="#FFFF66"]
"whoOwns"  [color="#FFFF66"]

{rank=same;
  "checkAddBackSlashes"
  "checkAddMacros"
  "checkAddPeriod"
  "checkAddSpaceSegments"
  "checkAddSpaces"
  "checkAlphabetic"
  "checkIeEgfun"
  "checkIsValidType"
  "checkLookForLeftBrace"
  "checkLookForRightBrace"
  "checkNumOfArgs"
  "checkSayBracket"
  "checkSkipBlanks"
  "checkSplitBackslash"
  "checkSplitOn"
  "checkSplitPunctuation"
  "firstNonBlankPosition"
  "getMatchingRightPren"
  "hasNoVowels"
  "htcharPosition"
  "newWordFrom"
  "removeBackslashes"
  "whoOwns"
}

"checkAddIndented"  [color="#22EE22"]
"checkDocMessage"  [color="#22EE22"]
"checkExtract"  [color="#22EE22"]
"checkGetArgs"  [color="#22EE22"]
"checkGetMargin"  [color="#22EE22"]
"checkGetParse"  [color="#22EE22"]
"checkGetStringBeforeRightBrace" [color="#22EE22"]
"checkIeEg"  [color="#22EE22"]
"checkIndentedLines"  [color="#22EE22"]
"checkSkipIdentifierToken"  [color="#22EE22"]
"checkSkipOpToken"  [color="#22EE22"]
"checkSplitBrace"  [color="#22EE22"]
"checkTrimCommented"  [color="#22EE22"]
"newString2Words"  [color="#22EE22"]

{rank=same;
  "checkAddIndented"
  "checkDocMessage"
  "checkExtract"
  "checkGetArgs"
  "checkGetMargin"
  "checkGetParse"
  "checkGetStringBeforeRightBrace"
```

```
  "checkIeEg"
  "checkIndentedLines"
  "checkSkipIdentifierToken"
  "checkSkipOpToken"
  "checkSplitBrace"
  "checkTrimCommented"
  "newString2Words"
}

"checkDocError"  [color="#2222DD"]
"checkRemoveComments"  [color="#2222DD"]
"checkSkipToken"  [color="#2222DD"]
"checkSplit2Words"  [color="#2222DD"]

{rank=same;
  "checkDocError"
  "checkRemoveComments"
  "checkSkipToken"
  "checkSplit2Words"
}

"checkBeginEnd"  [color="#FF6600"]
"checkDecorate"  [color="#FF6600"]
"checkDecorateForHt"  [color="#FF6600"]
"checkDocError1"  [color="#FF6600"]
"checkFixCommonProblem"  [color="#FF6600"]
"checkGetLispFunctionName"      [color="#FF6600"]
"checkHTargs"  [color="#FF6600"]
"checkRecordHash"  [color="#FF6600"]
"checkTexht"  [color="#FF6600"]
"checkTransformFirsts"  [color="#FF6600"]
"checkTrim"  [color="#FF6600"]

{rank=same;
  "checkBeginEnd"
  "checkDecorate"
  "checkDecorateForHt"
  "checkDocError1"
  "checkFixCommonProblem"
  "checkGetLispFunctionName"
  "checkHTargs"
  "checkRecordHash"
  "checkTexht"
  "checkTransformFirsts"
  "checkTrim"
}

"checkArguments"  [color="#0066FF"]
"checkBalance"  [color="#0066FF"]

{rank=same;
  "checkArguments"
  "checkBalance"
}
```

```
"checkComments"  [color="#006600"]
"checkRewrite"  [color="#006600"]

{rank=same;
  "checkComments"
  "checkRewrite"
}

"transformAndRecheckComments"  [color="#448822"]

"transDoc"  [color="#448822"]

"transDocList"                  [color="#448822"]

"finalizeDocumentation"  [color="#448822"]

"checkAddIndented"  ->  "firstNonBlankPosition"
"checkAddIndented"  ->  "checkAddSpaceSegments"
"checkArguments"  ->  "checkHTargs"
"checkBalance"  ->  "checkBeginEnd"
"checkBalance"  ->  "checkDocError"
"checkBalance"  ->  "checkSayBracket"
"checkBeginEnd"  ->  "checkDocError"
"checkComments" ->  "checkGetMargin"
"checkComments" ->  "checkTransformFirsts"
"checkComments" ->  "checkIndentedLines"
"checkComments" ->  "checkGetArgs"
"checkComments" ->  "newString2Words"
"checkComments" ->  "checkAddSpaces"
"checkComments" ->  "checkIeEg"
"checkComments" ->  "checkSplit2Words"
"checkComments" ->  "checkBalance"
"checkComments" ->  "checkArguments"
"checkComments" ->  "checkFixCommonProblem"
"checkComments" ->  "checkDecorate"
"checkComments" ->  "checkAddPeriod"
"checkDecorate"  ->  "checkDocError"
"checkDecorate"  ->  "checkAddBackSlashes"
"checkDecorate"  ->  "hasNoVowels"
"checkDecorateForHt"  ->  "checkDocError"
"checkDocError"  ->  "checkDocMessage"
"checkDocError1"  ->  "checkDocError"
"checkDocMessage"  ->  "whoOwns"
"checkExtract"  ->  "firstNonBlankPosition"
"checkFixCommonProblem"  ->  "checkDocError"
"checkGetArgs"  ->  "firstNonBlankPosition"
"checkGetArgs"  ->  "getMatchingRightPren"
"checkGetLispFunctionName"      ->  "checkDocError"
"checkGetMargin"  ->  "firstNonBlankPosition"
"checkGetParse"  ->  "removeBackslashes"
"checkHTargs"  ->  "checkLookForLeftBrace"
"checkHTargs"  ->  "checkLookForRightBrace"
"checkHTargs"  ->  "checkDocError"
```

```
"checkIeEg"  ->  "checkIeEgfun"
"checkIndentedLines"  ->  "firstNonBlankPosition"
"checkIndentedLines"  ->  "checkAddSpaceSegments"
"checkRecordHash"  ->  "checkLookForLeftBrace"
"checkRecordHash"  ->  "checkLookForRightBrace"
"checkRecordHash"  ->  "checkGetLispFunctionName"
"checkRecordHash"  ->  "checkGetStringBeforeRightBrace"
"checkRecordHash"  ->  "checkGetParse"
"checkRecordHash"  ->  "checkDocError"
"checkRecordHash"  ->  "checkNumOfArgs"
"checkRecordHash"  ->  "checkIsValidType"
"checkRemoveComments"  ->  "checkTrimCommented"
"checkRewrite"  ->  "checkRemoveComments"
"checkRewrite"  ->  "checkAddIndented"
"checkRewrite"  ->  "checkGetArgs"
"checkRewrite"  ->  "newString2Words"
"checkRewrite"  ->  "checkAddSpaces"
"checkRewrite"  ->  "checkSplit2Words"
"checkRewrite"  ->  "checkAddMacros"
"checkRewrite"  ->  "checkTexht"
"checkRewrite"  ->  "checkArguments"
"checkRewrite"  ->  "checkFixCommonProblem"
"checkRewrite"  ->  "checkRecordHash"
"checkRewrite"  ->  "checkDecorateForHt"
"checkSkipIdentifierToken"  ->  "checkAlphabetic"
"checkSkipOpToken"  ->  "checkAlphabetic"
"checkSkipToken"  ->  "checkSkipIdentifierToken"
"checkSkipToken"  ->  "checkSkipOpToken"
"checkSplit2Words"  ->  "checkSplitBrace"
"checkSplitBrace"  ->  "checkSplitBackslash"
"checkSplitBrace"  ->  "checkSplitOn"
"checkSplitBrace"  ->  "checkSplitPunctuation"
"checkTexht"  ->  "checkDocError"
"checkTransformFirsts"  ->  "checkSkipToken"
"checkTransformFirsts"  ->  "checkSkipBlanks"
"checkTransformFirsts"  ->  "getMatchingRightPren"
"checkTransformFirsts"  ->  "checkDocError"
"checkTrim"  ->  "checkDocError"
"checkTrimCommented"  ->  "htcharPosition"
"finalizeDocumentation"  ->  "transDocList"
"newString2Words"  ->  "newWordFrom"
"transDoc"  ->  "checkDocError1"
"transDoc"  ->  "checkTrim"
"transDoc"  ->  "checkExtract"
"transDoc"  ->  "transformAndRecheckComments"
"transDocList"  ->  "transDoc"
"transDocList"  ->  "checkDocError"
"transDocList"  ->  "checkDocError1"
"transformAndRecheckComments"   ->  "checkComments"
"transformAndRecheckComments"   ->  "checkRewrite"

"compileDocumentation"          ->  "finalizeDocumentation"
"finalizeLisplib"               ->  "finalizeDocumentation"
}
```

——–

12.1 Comment Checking Layer 0 – API

12.1.1 defun finalizeDocumentation

[bright p??]
[sayMSG p??]
[strconc p??]
[sayKeyedMsg p??]
[form2String p??]
[formatOpSignature p??]
[transDocList p446]
[assocleft p??]
[remdup p??]
[macroExpand p168]
[sublislis p??]
[$e p??]
[$lisplibForm p??]
[$docList p??]
[$op p??]
[$comblocklist p498]
[$FormalMapVariableList p249]

— **defun finalizeDocumentation** —

```
(defun |finalizeDocumentation| ()
 (labels (
  (fn (x env)
   (declare (special |$lisplibForm| |$FormalMapVariableList|))
   (cond
    ((atom x) (list x nil))
    (t
     (when (> (|#| x) 2) (setq x (take 2 x)))
     (sublislis |$FormalMapVariableList| (cdr |$lisplibForm|)
                   (|macroExpand| x env)))))
  (hn (u)
    ; ((op,sig,doc), ...)  --> ((op ((sig doc) ...)) ...)
   (let (opList op1 sig doc)
    (setq oplist (remdup (assocleft u)))
    (loop for op in opList
     collect
      (cons op
       (loop for item in u
        do (setq op1 (first item))
           (setq sig (second item))
           (setq doc (third item))
        when (equal op op1)
        collect
          (list sig doc)))))))
 (let (unusedCommentLineNumbers docList u noHeading attributes
```

```
      signatures name bigcnt op s litcnt a n r sig)
 (declare (special |$e| |$lisplibForm| |$docList| |$op| $comblocklist))
  (setq unusedCommentLineNumbers
   (loop for x in $comblocklist
    when (cdr x)
    collect x))
  (setq docList (subst '$ '% (|transDocList| |$op| |$docList|) :test #'equal))
  (cond
   ((setq u
     (loop for item in docList
      when (null (cdr item))
      collect (car item)))
    (loop for y in u
     do
      (cond
       ((eq y '|constructor|) (setq noHeading t))
       ((and (consp y) (consp (qcdr y)) (eq (qcddr y) nil)
             (consp (qcadr y)) (eq (qcaadr y) '|attribute|))
         (setq attributes (cons (cons (qcar y) (qcdadr y)) attributes)))
        (t (setq signatures (cons y signatures)))))
    (setq name (CAR |$lisplibForm|))
    (when (or noHeading signatures attributes unusedCommentLineNumbers)
      (|sayKeyedMsg| "Constructor documentation warnings (++ comments):" nil)
      (setq bigcnt 1)
      (when (or noHeading signatures attributes)
       (|sayKeyedMsg|
         " %1 The constructor %2 has missing documentation."
         (list (strconc (princ-to-string bigcnt) ".") name))
       (setq bigcnt (1+ bigcnt))
       (setq litcnt 1)
       (when noHeading
         (|sayKeyedMsg|
          " %x3 %1 The constructor %2 is missing the heading description."
          (list (strconc "(" (princ-to-string litcnt) ")") name))
         (setq litcnt (1+ litcnt)))
       (when signatures
         (|sayKeyedMsg|
          " %x3 %1 The following functions do not have documentation:"
          (list (strconc "(" (princ-to-string litcnt) ")")))
         (setq litcnt (1+ litcnt))
         (loop for item in signatures
          do
           (setq op (first item))
           (setq sig (second item))
           (setq s (|formatOpSignature| op sig))
           (|sayMSG|
            (if (atom s)
              (list '|%x9| s)
              (cons '|%x9| s)))))
       (when attributes
         (|sayKeyedMsg|
          " %x3 %1 The following attributes do not have documentation:"
          (list (strconc "(" (princ-to-string litcnt) ")")))
         (setq litcnt (1+ litcnt))
```

```
                            (DO ((G166491 attributes
                                     (CDR G166491))
                                 (x NIL))
                                ((OR (ATOM G166491)
                                     (PROGN
                                       (SETQ x (CAR G166491))
                                       NIL))
                                 NIL)
                              (SEQ (EXIT
                                    (PROGN
                                      (setq a (|form2String| x))
                                      (|sayMSG|
                                       (COND
                                         ((ATOM a)
                                          (CONS '|%x9| (CONS a NIL)))
                                         ('T (CONS '|%x9| a))))))))))
        (when unusedCommentLineNumbers
         (|sayKeyedMsg|
          " %1 The constructor %2 has incorrectly placed documentation."
          (list (strconc (princ-to-string bigcnt) ".") name))
         (loop for item in unusedCommentLineNumbers
          do
           (setq r (second item))
           (|sayMSG| (cons "   " (append (|bright| n) (list "   " r))))))))
  (hn
   (loop for item in docList
    collect (append (fn (car item) |$e|) (cdr item))))))
```

———-

12.2 Comment Checking Layer 1

12.2.1 defun transDocList

[sayBrightly p??]
[transDoc p447]
[checkDocError p467]
[checkDocError1 p457]
[$constructorName p??]

— defun transDocList —

```
(defun |transDocList| (|$constructorName| doclist)
 (declare (special |$constructorName|))
 (let (commentList conEntry acc)
  (|sayBrightly|
   (list "   Processing " |$constructorName| " for Browser database:"))
  (setq commentList (|transDoc| |$constructorName| doclist))
  (setq acc nil)
  (loop for entry in commentList
   do
    (cond
```

```
    ((and (consp entry) (eq (qcar entry) '|constructor|)
          (consp (qcdr entry)) (eq (qcddr entry) nil))
      (if conEntry
       (|checkDocError| (list "Spurious comments: " (qcadr entry)))
       (setq conEntry entry)))
    (t (setq acc (cons entry acc)))))
 (if conEntry
  (cons conEntry acc)
  (progn
   (|checkDocError1| (list "Missing Description"))
   acc))))
```

———-

12.3 Comment Checking Layer 2

12.3.1 defun transDoc

[checkDocError1 p457]
[checkTrim p466]
[checkExtract p469]
[transformAndRecheckComments p448]
[nreverse p??]
[$x p??]
[$attribute? p??]
[$x p??]
[$attribute? p??]
[$argl p??]

— defun transDoc —

```
(defun |transDoc| (conname doclist)
 (declare (ignore conname))
 (let (|$x| |$attribute?| |$argl| rlist lines u v longline acc)
 (declare (special |$x| |$attribute?| |$argl|))
  (setq |$x| nil)
  (setq rlist (reverse doclist))
  (loop for item in rlist
   do
    (setq |$x| (car item))
    (setq lines (cdr item))
    (setq |$attribute?|
      (and (consp |$x|) (consp (qcdr |$x|)) (eq (qcddr |$x|) nil)
           (consp (qcadr |$x|)) (eq (qcdadr |$x|) nil)
           (eq (qcaadr |$x|) '|attribute|)))
    (cond
     ((null lines)
      (unless |$attribute?| (|checkDocError1| (list "Not documented!!!!"))))
     (t
      (setq u
       (|checkTrim| |$x|
```

```
        (cond
         ((stringp lines) (list lines))
         ((eq |$x| '|constructor|) (car lines))
         (t lines))))
      (setq |$argl| nil) ;; possibly unused -- tpd
      (setq longline
       (cond
        ((eq |$x| '|constructor|)
          (setq v
           (or
            (|checkExtract| "Description:" u)
            (and u (|checkExtract| "Description:"
                    (cons (strconc "Description: " (car u)) (cdr u))))))
          (|transformAndRecheckComments| '|constructor| (or v u)))
        (t (|transformAndRecheckComments| |$x| u))))
      (setq acc (cons (list |$x| longline) acc)))))
    (nreverse acc)))
```

———

12.4 Comment Checking Layer 3

12.4.1 defun transformAndRecheckComments

[sayBrightly p??]
[checkComments p449]
[checkRewrite p450]
[$exposeFlagHeading p??]
[$checkingXmptex? p??]
[$x p??]
[$name p??]
[$origin p??]
[$recheckingFlag p??]
[$exposeFlagHeading p??]

— defun transformAndRecheckComments —

```
(defun |transformAndRecheckComments| (name lines)
 (let (|$x| |$name| |$origin| |$recheckingFlag| |$exposeFlagHeading| u)
 (declare (special |$x| |$name| |$origin| |$recheckingFlag|
                   |$exposeFlagHeading| |$exposeFlag| |$checkingXmptex?|))
  (setq |$checkingXmptex?| nil)
  (setq |$x| name)
  (setq |$name| '|GlossaryPage|)
  (setq |$origin| '|gloss|)
  (setq |$recheckingFlag| nil)
  (setq |$exposeFlagHeading| (list "--------" name "---------"))
  (unless |$exposeFlag| (|sayBrightly| |$exposeFlagHeading|))
  (setq u (|checkComments| name lines))
  (setq |$recheckingFlag| t)
  (|checkRewrite| name (list u))
```

```
  (setq |$recheckingFlag| nil)
  u))
```

————-

12.5 Comment Checking Layer 4

12.5.1 defun checkComments

[checkGetMargin p471]
[checkTransformFirsts p463]
[checkIndentedLines p473]
[checkGetArgs p470]
[newString2Words p475]
[checkAddSpaces p478]
[checkIeEg p472]
[checkSplit2Words p468]
[checkBalance p452]
[checkArguments p451]
[checkFixCommonProblems p??]
[checkDecorate p454]
[strconc p??]
[checkAddPeriod p477]
[pp p??]
[$attribute? p??]
[$checkErrorFlag p??]
[$argl p??]
[$checkErrorFlag p??]

— defun checkComments —

```
(defun |checkComments| (nameSig lines)
 (let (|$checkErrorFlag| margin w verbatim u2 okBefore u v res)
 (declare (special |$checkErrorFlag| |$argl| |$attribute?|))
  (setq |$checkErrorFlag| nil)
  (setq margin (|checkGetMargin| lines))
  (cond
   ((and (or (null (boundp '|$attribute?|)) (null |$attribute?|))
         (not (eq nameSig '|constructor|)))
     (setq lines
      (cons
       (|checkTransformFirsts| (car nameSig) (car lines) margin)
       (cdr lines))))
     (setq u (|checkIndentedLines| lines margin))
     (setq |$argl| (|checkGetArgs| (car u)))
     (setq u2 nil)
     (setq verbatim nil)
     (loop for x in u
      do (setq w (|newString2Words| x))
         (cond
```

```
          (verbatim
           (cond
            ((and w (equal (car w) "\\end{verbatim}"))
              (setq verbatim nil)
              (setq u2 (append u2 w)))
            (t
             (setq u2 (append u2 (list x))))))
          ((and w (equal (car w) "\\begin{verbatim}"))
            (setq verbatim t)
            (setq u2 (append u2 w)))
          (t (setq u2 (append u2 w)))))
     (setq u u2)
     (setq u (|checkAddSpaces| u))
     (setq u (|checkIeEg| u))
     (setq u (|checkSplit2Words| u))
     (|checkBalance| u)
     (setq okBefore (null |$checkErrorFlag|))
     (|checkArguments| u)
     (when |$checkErrorFlag| (setq u (|checkFixCommonProblem| u)))
     (setq v (|checkDecorate| u))
     (setq res
      (let ((result ""))
       (loop for y in v
        do (setq result (strconc result y)))
      result))
     (setq res (|checkAddPeriod| res))
     (when |$checkErrorFlag| (|pp| res))
     res))
```

———-

12.5.2 defun checkRewrite

[checkRemoveComments p467]
[checkAddIndented p469]
[checkGetArgs p470]
[newString2Words p475]
[checkAddSpaces p478]
[checkSplit2Words p468]
[checkAddMacros p477]
[checkTexht p462]
[checkArguments p451]
[checkFixCommonProblem p457]
[checkRecordHash p459]
[checkDecorateForHt p456]
[$checkErrorFlag p**??**]
[$argl p**??**]
[$checkingXmptex? p**??**]

— **defun checkRewrite** —

```
(defun |checkRewrite| (name lines)
```

```
 (declare (ignore name))
 (prog (|$checkErrorFlag| margin w verbatim u2 okBefore u)
 (declare (special |$checkErrorFlag| |$argl| |$checkingXmptex?|))
  (setq |$checkErrorFlag| t)
  (setq margin 0)
  (setq lines (|checkRemoveComments| lines))
  (setq u lines)
  (when |$checkingXmptex?|
   (setq u
    (loop for x in u
     collect (|checkAddIndented| x margin))))
  (setq |$argl| (|checkGetArgs| (car u)))
  (setq u2 nil)
  (setq verbatim nil)
  (loop for x in u
   do
    (setq w (|newString2Words| x))
    (cond
     (verbatim
      (cond
       ((and w (equal (car w) "\\end{verbatim}"))
         (setq verbatim nil)
         (setq u2 (append u2 w)))
       (t
         (setq u2 (append u2 (list x))))))
     ((and w (equal (car w) "\\begin{verbatim}"))
      (setq verbatim t)
      (setq u2 (append u2 w)))
     (t (setq u2 (append u2 w)))))
  (setq u u2)
  (setq u (|checkAddSpaces| u))
  (setq u (|checkSplit2Words| u))
  (setq u (|checkAddMacros| u))
  (setq u (|checkTexht| u))
  (setq okBefore (null |$checkErrorFlag|))
  (|checkArguments| u)
  (when |$checkErrorFlag| (setq u (|checkFixCommonProblem| u)))
  (|checkRecordHash| u)
  (|checkDecorateForHt| u)))
```

———-

12.6 Comment Checking Layer 5

12.6.1 defun checkArguments

[hget p??]
[checkHTargs p458]
[$htMacroTable p??]

— defun checkArguments —

```
(defun |checkArguments| (u)
 (let (x k)
 (declare (special |$htMacroTable|))
  (loop while u
   do (setq x (car u))
      (cond
       ((null (setq k (hget |$htMacroTable| x))) '|skip|)
       ((eql k 0) '|skip|)
       ((> k 0) (|checkHTargs| x (cdr u) k nil))
       (t (|checkHTargs| x (cdr u) (- k) t)))
       (pop u))
  u))
```

———-

12.6.2 defun checkBalance

[checkBeginEnd p453]
[assoc p??]
[rassoc p??]
[checkDocError p467]
[checkSayBracket p482]
[nreverse p??]
[$checkPrenAlist p??]

— defun checkBalance —

```
(defun |checkBalance| (u)
 (let (x openClose open top restStack stack)
 (declare (special |$checkPrenAlist|))
  (|checkBeginEnd| u)
  (setq stack nil)
  (loop while u
   do
    (setq x (car u))
    (cond
     ((setq openClose (|assoc| x |$checkPrenAlist|))
       (setq stack (cons (car openClose) stack)))
     ((setq open (|rassoc| x |$checkPrenAlist|))
       (cond
        ((consp stack)
          (setq top (qcar stack))
          (setq restStack (qcdr stack))
          (when (not (eq open top))
            (|checkDocError|
              (list "Mismatch: left "  (|checkSayBracket| top)
                    " matches right " (|checkSayBracket| open))))
          (setq stack restStack))
        (t
         (|checkDocError|
          (list "Missing left " (|checkSayBracket| open)))))))
    (pop u))
```

```
  (when stack
   (loop for x in (nreverse stack)
    do
     (|checkDocError| (list "Missing right " (|checkSayBracket| x)))))
 u))
```

———-

12.7 Comment Checking Layer 6

12.7.1 defun checkBeginEnd

[length p??]
[hget p??]
[ifcar p??]
[ifcdr p??]
[substring? p??]
[checkDocError p467]
[member p??]
[$beginEndList p??]
[$htMacroTable p??]

— **defun checkBeginEnd** —

```
(defun |checkBeginEnd| (u)
 (let (x y beginEndStack)
 (declare (special |$beginEndList| |$htMacroTable|))
  (loop while u
   do
    (setq x (car u))
    (cond
     ((and (stringp x) (equal (elt x 0) #\\) (> (|#| x) 2)
           (null (hget |$htMacroTable| x)) (null (equal x "\\spadignore"))
           (equal (ifcar (ifcdr u)) #\{)
           (null (or (|substring?| "\\radiobox" x 0)
                     (|substring?| "\\inputbox" x 0))))
       (|checkDocError| (list '|Unexpected HT command: | x)))
     ((equal x "\\beginitems")
      (setq beginEndStack (cons '|items| beginEndStack)))
     ((equal x "\\begin")
      (cond
       ((and (consp u) (consp (qcdr u)) (equal (qcar (qcdr u)) #\{)
             (consp (qcddr u)) (equal (car (qcdddr u)) #\}))
         (setq y (qcaddr u))
         (cond
          ((null (|member| y |$beginEndList|))
            (|checkDocError| (list "Unknown begin type: \\begin{" y "}"))))
         (setq beginEndStack (cons y beginEndStack))
         (setq u (qcdddr u)))
       (t (|checkDocError| (list "Improper \\begin command")))))
     ((equal x "\\item")
```

```
      (cond
       ((|member| (ifcar beginEndStack) '("items" "menu")) nil)
       ((null beginEndStack)
         (|checkDocError| (list "\\item appears outside a \\begin-\\end")))
       (t
         (|checkDocError|
          (list "\\item appears within a \\begin{"
                (ifcar beginEndStack) "}..")))))
     ((equal x "\\end")
      (cond
       ((and (consp u) (consp (qcdr u)) (equal (qcar (qcdr u)) #\{)
             (consp (qcddr u)) (equal (car (qcdddr u)) #\}))
         (setq y (qcaddr u))
         (cond
          ((equal y (ifcar beginEndStack))
            (setq beginEndStack (cdr beginEndStack))
            (setq u (qcdddr u)))
          (t
           (|checkDocError|
            (list "Trying to match \\begin{" (ifcar beginEndStack)
                  "} with \\end{" y "}")))))
        (t
         (|checkDocError| (list "Improper \\end command"))))))
   (pop u))
   (cond
    (beginEndStack
      (|checkDocError| (list "Missing \\end{" (car beginEndStack) "}")))
    (t '|ok|))))
```

———-

12.7.2 defun checkDecorate

[checkDocError p467]
[member p??]
[checkAddBackSlashes p476]
[hasNoVowels p486]
[$checkingXmptex? p??]
[$charExclusions p??]
[$argl p??]

— defun checkDecorate —

```
(defun |checkDecorate| (u)
 (let (x count mathSymbolsOk spadflag verbatim v xcount acc)
 (declare (special |$argl| |$charExclusions| |$checkingXmptex?|))
  (setq count 0)
  (loop while u
   do
    (setq x (car u))
    (cond
     ((null verbatim)
```

```
    (cond
     ((string= x "\\em")
       (cond
        ((> count 0)
          (setq mathSymbolsOk (1- count))
          (setq spadflag (1- count)))
        (t
         (|checkDocError| (list "\\em must be enclosed in braces"))))))
    (when (|member| x '("\\spadpaste" "\\spad" "\\spadop"))
      (setq mathSymbolsOk count))
    (cond
     ((|member| x '("\\s" "\\spadtype" "\\spadsys" "\\example" "\\andexample"
                    "\\spadop" "\\spad" "\\spadignore" "\\spadpaste"
                    "\\spadcommand" "\\footnote"))
        (setq spadflag count))
     ((equal x #\{)
        (setq count (1+ count)))
     ((equal x #\})
        (setq count (1- count))
        (when (eql mathSymbolsOk count) (setq mathSymbolsOk nil))
        (when (eql spadflag count) (setq spadflag nil)))
     ((and (null mathSymbolsOk)
           (|member| x '("+" "*" "=" "==" "->")))
       (when |$checkingXmptex?|
         (|checkDocError|
          (list '|Symbol | x " appearing outside \\spad{}")))))))
  (setq acc
   (cond
    ((string= x "\\end{verbatim}")
      (setq verbatim nil)
      (cons x acc))
    (verbatim (cons x acc))
    ((string= x "\\begin{verbatim}")
     (setq verbatim t)
     (cons x acc))
    ((and (string= x "\\begin")
          (equal (car (setq v (ifcdr u))) #\{)
          (string= (car (setq v (ifcdr v))) "detail")
          (equal (car (setq v (ifcdr v))) #\}))
      (setq u v)
      (cons "\\blankline " acc))
    ((and (string= x "\\end")
          (equal (car (setq v (ifcdr u))) #\{)
          (string= (car (setq v (ifcdr v))) "detail")
          (equal (car (setq v (ifcdr v))) #\}))
      (setq u v)
      acc)
    ((or (char= x #\$) (string= x "$"))
      (cons "\\$" acc))
    ((or (char= x #\%) (string= x "%"))
      (cons "\\%" acc))
    ((or (char= x #\,) (string= x ","))
      (cons ",{}" acc))
    ((string= x "\\spad")
```

```
      (cons "\\spad" acc))
    ((and (stringp x) (digitp (elt x 0)))
      (cons x acc))
    ((and (null spadflag)
          (or (and (charp x)
                   (alpha-char-p x)
                   (null (member x |$charExclusions|)))
              (|member| x |$argl|)))
      (cons #\} (cons x (cons #\{ (cons "\\spad" acc)))))
    ((and (null spadflag)
          (or (and (stringp x)
                   (null (equal (elt x 0) #\\))
                   (digitp (elt x (maxindex x))))
              (|member| x '("true" "false"))))
      (cons #\} (cons x (cons #\{ (cons "\\spad" acc)))))
    (t
     (setq xcount (|#| x))
     (cond
      ((and (eql xcount 3)
            (char= (elt x 1) #\t)
            (char= (elt x 2) #\h))
        (cons "th" (cons #\}
          (cons (elt x 0) (cons #\{ (cons "\\spad" acc))))))
      ((and (eql xcount 4)
            (char= (elt x 1) #\-)
            (char= (elt x 2) #\t)
            (char= (elt x 3) #\h))
       (cons "-th" (cons #\}
         (cons (elt x 0) (cons #\{ (cons "\\spad" acc))))))
      ((or (and (eql xcount 2)
                (char= (elt x 1) #\i))
           (and (null spadflag)
                (> xcount 0)
                (> 4 xcount)
                (null (|member| x '("th" "rd" "st")))
                (|hasNoVowels| x)))
       (cons #\}
         (cons x (cons #\{ (cons "\\spad" acc)))))
      (t
       (cons (|checkAddBackSlashes| x) acc))))))
   (setq u (cdr u)))
  (nreverse acc)))
```

———-

12.7.3 defun checkDecorateForHt

[checkDocError p467]
[member p??]
[$checkingXmptex? p??]

— defun checkDecorateForHt —

```
(defun |checkDecorateForHt| (u)
 (let (x count spadflag)
 (declare (special |$checkingXmptex?|))
  (setq count 0)
  (setq spadflag nil)
  (loop while u
   do
   (setq x (car u))
   (when (equal x "\\em")
    (if (> count 0)
     (setq spadflag (1- count))
     (|checkDocError| (list "\\em must be enclosed in braces"))))
   (cond
    ((|member| x '("\\s" "\\spadop" "\\spadtype" "\\spad" "\\spadpaste"
                   "\\spadcommand" "\\footnote"))
     (setq spadflag count))
    ((equal x #\{)
     (setq count (1+ count)))
    ((equal x #\})
     (setq count (1- count))
     (when (equal spadflag count) (setq spadflag nil)))
    ((and (null spadflag) (|member| x '("+" "*" "=" "==" "->")))
     (when |$checkingXmptex?|
       (|checkDocError| (list '|Symbol | x " appearing outside \\spad{}"))))
    (t nil))
   (when (or (equal x "$") (equal x "%"))
      (|checkDocError| (list "Unescaped " x)))
   (pop u))
  u))
```

———-

12.7.4 defun checkDocError1

[checkDocError p467]
[$compileDocumentation p160]

— defun checkDocError1 —

```
(defun |checkDocError1| (u)
 (declare (special |$compileDocumentation|))
 (if (and (boundp '|$compileDocumentation|) |$compileDocumentation|)
  nil
  (|checkDocError| u)))
```

———-

12.7.5 defun checkFixCommonProblem

[member p??]
[ifcar p??]

[ifcdr p??]
[checkDocError p467]
[$HTspadmacros p??]

— defun checkFixCommonProblem —

```
(defun |checkFixCommonProblem| (u)
 (let (x next acc)
 (declare (special |$HTspadmacros|))
  (loop while u
   do
    (setq x (car u))
    (cond
     ((and (equal x #\{)
           (|member| (setq next (ifcar (cdr u))) |$HTspadmacros|)
           (not (equal (ifcar (ifcdr (cdr u))) #\{)))
       (|checkDocError| (list "Reversing " next " and left brace"))
       (setq acc (cons #\{ (cons next acc)))
       (setq u (cddr u)))
     (t
      (setq acc (cons x acc))
      (setq u (cdr u)))))
  (nreverse acc)))
```

———-

12.7.6 defun checkGetLispFunctionName

[charPosition p??]
[checkDocError p467]

— defun checkGetLispFunctionName —

```
(defun |checkGetLispFunctionName| (s)
 (let (n k j)
  (setq n (|#| s))
  (cond
   ((and (setq k (|charPosition| #\| s 1))
         (> n k)
         (setq j (|charPosition| #\| s (1+ k)))
         (> n j))
     (substring s (1+ k) (1- (- j k))))
   (t
    (|checkDocError| (cons "Ill-formed lisp expression : " (list s)))
    '|illformed|))))
```

———-

12.7.7 defun checkHTargs

Note that `u` should start with an open brace. [checkLookForLeftBrace p481]
[checkLookForRightBrace p481]

[checkDocError p467]
[checkHTargs p458]
[ifcdr p??]

— defun checkHTargs —

```
(defun |checkHTargs| (keyword u nargs inteerValue?)
  (cond
   ((eql nargs 0) '|ok|)
   ((null (setq u (|checkLookForLeftBrace| u)))
      (|checkDocError| (list "Missing argument for " keyword)))
   ((null (setq u (|checkLookForRightBrace| (ifcdr u))))
      (|checkDocError| (list "Missing right brace for " keyword)))
    (t
     (|checkHTargs| keyword (cdr u) (1- nargs) inteerValue?))))
```

———-

12.7.8 defun checkRecordHash

[member p??]
[checkLookForLeftBrace p481]
[checkLookForRightBrace p481]
[ifcdr p??]
[intern p??]
[hget p??]
[hput p??]
[checkGetLispFunctionName p458]
[checkGetStringBeforeRightBrace p472]
[checkGetParse p471]
[checkDocError p467]
[opOf p??]
[spadSysChoose p461]
[checkNumOfArgs p481]
[checkIsValidType p480]
[form2HtString p??]
[getl p??]
[$HTlinks p??]
[$htHash p??]
[$HTlisplinks p??]
[$lispHash p??]
[$glossHash p??]
[$currentSysList p??]
[$setOptions p??]
[$sysHash p??]
[$name p??]
[$origin p??]
[$sysHash p??]
[$glossHash p??]
[$lispHash p??]

[$htHash p??]

— defun checkRecordHash —

```
(defun |checkRecordHash| (u)
 (let (p q htname entry s parse n key x)
 (declare (special |$origin| |$name| |$sysHash| |$setOptions| |$glossHash|
                   |$currentSysList| |$lispHash| |$HTlisplinks| |$htHash|
                   |$HTlinks|))
  (loop while u
   do
   (setq x (car u))
   (when (and (stringp x) (equal (elt x 0) #\\))
    (cond
     ((and (|member| x |$HTlinks|)
           (setq u (|checkLookForLeftBrace| (ifcdr u)))
           (setq u (|checkLookForRightBrace| (ifcdr u)))
           (setq u (|checkLookForLeftBrace|  (ifcdr u)))
           (setq u (ifcdr u)))
       (setq htname (|intern| (ifcar u)))
       (setq entry (or (hget |$htHash| htname) (list nil)))
       (hput |$htHash| htname
        (cons (car entry) (cons (cons |$name| |$origin|) (cdr entry)))))
     ((and (|member| x |$HTlisplinks|)
           (setq u (|checkLookForLeftBrace| (ifcdr u)))
           (setq u (|checkLookForRightBrace| (ifcdr u)))
           (setq u (|checkLookForLeftBrace| (ifcdr u)))
           (setq u (ifcdr u)))
       (setq htname
        (|intern|
         (|checkGetLispFunctionName|
          (|checkGetStringBeforeRightBrace| u))))
       (setq entry (or (hget |$lispHash| htname) (list nil)))
       (hput |$lispHash| htname
        (cons (car entry) (cons (cons |$name| |$origin|) (cdr entry)))))
     ((and (or (setq p (|member| x '("\\gloss" "\\spadglos")))
               (setq q (|member| x '("\\glossSee" "\\spadglosSee"))))
           (setq u (|checkLookForLeftBrace| (ifcdr u)))
           (setq u (ifcdr u)))
       (when q
        (setq u (|checkLookForRightBrace| u))
        (setq u (|checkLookForLeftBrace| (ifcdr u)))
        (setq u (ifcdr u)))
       (setq htname (|intern| (|checkGetStringBeforeRightBrace| u)))
       (setq entry
        (or (hget |$glossHash| htname) (list nil)))
            (hput |$glossHash| htname
             (cons (car entry) (cons (cons |$name| |$origin|) (cdr entry)))))
     ((and (boot-equal x "\\spadsys")
           (setq u (|checkLookForLeftBrace| (ifcdr u)))
           (setq u (ifcdr u)))
       (setq s (|checkGetStringBeforeRightBrace| u))
       (when (char= (elt s 0) #\)) (setq s (substring s 1 nil)))
       (setq parse (|checkGetParse| s))
```

```
      (cond
       ((null parse)
        (|checkDocError| (list "Unparseable \\spadtype: " s)))
       ((null (|member| (|opOf| parse) |$currentSysList|))
        (|checkDocError| (list "Bad system command: " s)))
       ((or (atom parse)
            (null (and (consp parse) (eq (qcar parse) '|set|)
                       (consp (qcdr parse))
                       (eq (qcddr parse) nil))))
        '|ok|)
       ((null (|spadSysChoose| |$setOptions| (qcadr parse)))
         (progn
          (|checkDocError| (list "Incorrect \\spadsys: " s))
          (setq entry (or (hget |$sysHash| htname) (list nil)))
          (hput |$sysHash| htname
           (cons (car entry) (cons (cons |$name| |$origin|) (cdr entry))))))))
    ((and (boot-equal x "\\spadtype")
          (setq u (|checkLookForLeftBrace| (ifcdr u)))
          (setq u (ifcdr u)))
      (setq s (|checkGetStringBeforeRightBrace| u))
      (setq parse (|checkGetParse| s))
      (cond
       ((null parse)
        (|checkDocError| (list "Unparseable \\spadtype: " s)))
       (t
        (setq n (|checkNumOfArgs| parse))
        (cond
         ((null n)
          (|checkDocError| (list "Unknown \\spadtype: " s)))
         ((and (atom parse) (> n 0))
          '|skip|)
         ((null (setq key (|checkIsValidType| parse)))
           (|checkDocError| (list "Unknown \\spadtype: " s)))
         ((atom key) '|ok|)
         (t
          (|checkDocError|
           (list "Wrong number of arguments: " (|form2HtString| key))))))))
    ((and (|member| x '("\\spadop" "\\keyword"))
          (setq u (|checkLookForLeftBrace| (ifcdr u)))
          (setq u (ifcdr u)))
     (setq x (|intern| (|checkGetStringBeforeRightBrace| u)))
     (when (null (or (getl x '|Led|) (getl x '|Nud|)))
        (|checkDocError| (list "Unknown \\spadop: " x))))))
  (pop u))
 '|done|))
```

———

12.7.9 defun spadSysChoose

[lassoc p??]
[spadSysBranch p462]

— defun spadSysChoose —

```
(defun |spadSysChoose| (tree form) ; tree is ((word . tree) ..)
 (let (lookupOn newTree)
  (cond
   ((null form) t)
   ((null tree) nil)
   (t
     (if (and (consp form) (consp (qcdr form)) (eq (qcddr form) nil))
       (setq lookupOn (qcar form))
       (setq lookupOn form))
    (when (setq newTree (lassoc lookupOn tree))
        (|spadSysBranch| newTree (cadr form)))))))
```

———-

12.7.10 defun spadSysBranch

[spadSysChoose p461]
[member p??]
[systemError p??]

— defun spadSysBranch —

```
(defun |spadSysBranch| (tree arg) ; tree is (msg kind TREEorSomethingElse ...)
 (let (kind)
  (cond
   ((null arg) t)
   (t
    (setq kind (elt tree 2))
    (cond
     ((eq kind 'tree) (|spadSysChoose| (elt tree 4) arg))
     ((eq kind 'literals) (|member| arg (ELT tree 4)))
     ((eq kind 'integer) (integerp arg))
     ((eq kind 'function) (atom arg))
     (t (|systemError| "unknown tree branch")))))))
```

———-

12.7.11 defun checkTexht

[ifcdr p??]
[ifcar p??]
[checkDocError p467]

— defun checkTexht —

```
(defun |checkTexht| (u)
 (let (count y x acc)
  (setq count 0)
  (loop while u
```

```
  do
   (setq x (car u))
   (when (and (string= x "\\texht") (setq u (ifcdr u)))
    (unless  (equal (ifcar u) #\{)
      (|checkDocError| "First left brace after \\texht missing"))
    ; drop first argument including braces of \texht
    (setq count 1)
    (loop while
           (or (not (equal (setq y (ifcar (setq u (cdr u)))) #\}))
               (> count 1))
     do
      (when (equal y #\{) (setq count (1+ count)))
      (when (equal y #\}) (setq count (1- count))))
    ; drop first right brace of 1st arg
    (setq x (ifcar (setq u (cdr u)))))
   (when (and (string= x "\\httex")
              (setq u (ifcdr u))
              (equal (ifcar u) #\{))
      ; left  brace: add it
      (setq acc (cons (ifcar u) acc))
      (loop while
             (not (equal (setq y (ifcar (setq u (cdr u)))) #\}))
        do (setq acc (cons y acc)))
      ; right brace: add it
      (setq acc (cons (ifcar u) acc))
      ; left brace:  forget it
      (setq x (ifcar (setq u (cdr u))))
      (loop while (not (equal (ifcar (setq u (cdr u))) #\}))
       do '|skip|)
      ; forget right brace: move to next char
      (setq x (ifcar (setq u (cdr u)))))
   (setq acc (cons x acc))
   (setq u (cdr u)))
 (nreverse acc)))
```

———-

12.7.12 defun checkTransformFirsts

[pname p??]
[leftTrim p??]
[fillerSpaces p??]
[checkTransformFirsts p463]
[maxindex p??]
[checkSkipToken p468]
[checkSkipBlanks p482]
[getMatchingRightPren p485]
[checkDocError p467]
[strconc p??]
[getl p??]
[lassoc p??]

[$checkPrenAlist p??]

— defun checkTransformFirsts —

```
(defun |checkTransformFirsts| (opname u margin)
 (prog (namestring s m infixOp p open close z n i prefixOp j k firstWord)
 (declare (special |$checkPrenAlist|))
  (return
   (progn
; case 1: \spad{...
; case 2: form(args)
    (setq u (string-trim '(#\space) u)) ; spaces confuse us
    (setq namestring (pname opname))
    (cond
     ((equal namestring "Zero") (setq namestring "0"))
     ((equal namestring "One")  (setq namestring "1"))
     (t nil))
    (cond
     ((> margin 0)
      (setq s (|leftTrim| u))
      (strconc (|fillerSpaces| margin) (|checkTransformFirsts| opname s 0)))
     (t
      (setq m (maxindex u))
      (cond
       ((> 2 m) u)
       ((equal (elt u 0) #\\) u)
       ((alpha-char-p (elt u 0))
         (setq i (or (|checkSkipToken| u 0 m) (return u)))
         (setq j (or (|checkSkipBlanks| u i m) (return u)))
         (setq open (elt u j))
         (cond
          ((or (and (equal open #\[) (setq close #\]))
               (and (equal open #\() (setq close #\))))
            (setq k (|getMatchingRightPren| u (1+ j) open close))
            (cond
             ((not (equal namestring (setq firstWord (substring u 0 i))))
               (|checkDocError|
                 (list "(1) Improper first word in comments: " firstWord))
                 u)
             ((null k)
              (cond
               ((equal open #\[)
                 (|checkDocError|
                   (list "Missing close bracket on first line: " u)))
               (t
                 (|checkDocError|
                   (list "Missing close parenthesis on first line: " u))))
                    u)
             (t
              (strconc "\\spad{" (substring u 0 (1+ k)) "}"
                                   (substring u (1+ k) nil)))))
          (t
           (setq k (or (|checkSkipToken| u j m) (return u)))
           (setq infixOp (intern (substring u j (- k j))))
```

```
          (cond
; case 3: form arg
           ((null (getl infixOp '|Led|))
            (cond
             ((not (equal namestring (setq firstWord (substring u 0 i))))
              (|checkDocError|
               (list "(2) Improper first word in comments: " firstWord))
              u)
             ((and (eql (|#| (setq p (pname infixOp))) 1)
               (setq open (elt p 0))
               (setq close (lassoc open |$checkPrenAlist|)))
              (setq z (|getMatchingRightPren| u (1+ k) open close))
              (when (> z (maxindex u)) (setq z (1- k)))
              (strconc "\\spad{" (substring u 0 (1+ z)) "}"
                                  (substring u (1+ z) nil)))
             (t
              (strconc "\\spad{" (substring u 0 k) "}"
                                  (substring u k nil)))))
           (t
            (setq z (or (|checkSkipBlanks| u k m) (return u)))
            (setq n (or (|checkSkipToken| u z m) (return u)))
            (cond
             ((not (equal namestring (pname infixOp)))
               (|checkDocError|
                (list "(3) Improper initial operator in comments: " infixOp))
               u)
             (t
               (strconc "\\spad{" (substring u 0 n) "}"
                                   (substring u n nil))))))))
; case 4: arg op arg
       (t
        (setq i (or (|checkSkipToken| u 0 m) (return u)))
        (cond
         ((not (equal namestring (setq firstWord (substring u 0 i))))
           (|checkDocError|
            (list "(4) Improper first word in comments: " firstWord))
           u)
         (t
           (setq prefixOp (intern (substring u 0 i)))
           (cond
            ((null (getl prefixOp '|Nud|)) u)
            (t
             (setq j (or (|checkSkipBlanks| u i m) (return u)))
             (cond
; case 5: op arg
              ((equal (elt u j) #\( )
                (setq j
                 (|getMatchingRightPren| u (1+ j) #\( #\) ))
                (cond
                 ((> j m) u)
                 (t
                  (strconc "\\spad{" (substring u 0 (1+ j)) "}"
                                      (substring u (1+ j) nil)))))
              (t
```

```
               (setq k (or (|checkSkipToken| u j m) (return u)))
               (cond
                ((not (equal namestring (setq firstWord (substring u 0 i))))
                  (|checkDocError|
                   (list "(5) Improper first word in comments: " firstWord))
                  u)
                (t
                 (strconc "\\spad{" (substring u 0 k) "}"
                                   (substring u k nil)))))))))))))))))
```

———-

12.7.13 defun checkTrim

[charPosition p??]
[systemError p??]
[checkDocError p467]
[$charBlank p??]
[$x p??]
[$charPlus p??]

— defun checkTrim —

```
(defun |checkTrim| (|$x| lines)
 (declare (special |$x|))
 (labels (
  (trim (s)
   (let (k)
    (setq k (wherePP s))
    (substring s (+ k 2) nil)))
  (wherePP (u)
   (let (k)
    (setq k (|charPosition| #\+ u 0))
    (if (or (eql k (|#| u))
            (not (eql (|charPosition| #\+ u (1+ k)) (1+ k))))
      (|systemError| " Improper comment found")
      k))))
 (let (j s)
  (setq s  (list (wherePP (car lines))))
  (loop for x in (rest lines)
   do
    (setq j (wherePP x))
    (unless (member j s)
      (|checkDocError| (list |$x| " has varying indentation levels"))
      (setq s (cons j s))))
  (loop for y in lines
   collect (trim y)))))
```

———-

12.8 Comment Checking Layer 7

12.8.1 defun checkDocError

[checkDocMessage p469]
[concat p??]
[saybrightly1 p??]
[sayBrightly p??]
[$checkErrorFlag p??]
[$recheckingFlag p??]
[$constructorName p??]
[$exposeFlag p??]
[$exposeFlagHeading p??]
[$outStream p??]
[$checkErrorFlag p??]
[$exposeFlagHeading p??]

— defun checkDocError —

```
(defun |checkDocError| (u)
 (let (msg)
 (declare (special |$outStream| |$exposeFlag| |$exposeFlagHeading|
                   |$constructorName| |$recheckingFlag| |$checkErrorFlag|))
  (setq |$checkErrorFlag| t)
  (setq msg
   (cond
    (|$recheckingFlag|
      (if |$constructorName|
        (|checkDocMessage| u)
        (|concat| "> " u)))
    (|$constructorName| (|checkDocMessage| u))
    (t u)))
  (when (and |$exposeFlag| |$exposeFlagHeading|)
     (saybrightly1 |$exposeFlagHeading| |$outStream|)
     (|sayBrightly| |$exposeFlagHeading|)
     (setq |$exposeFlagHeading| nil))
  (|sayBrightly| msg)
  (when |$exposeFlag| (saybrightly1 msg |$outStream|))))
```

———-

12.8.2 defun checkRemoveComments

[checkTrimCommented p475]

— defun checkRemoveComments —

```
(defun |checkRemoveComments| (lines)
 (let (line acc)
  (loop while lines
   do
   (setq line (|checkTrimCommented| (car lines)))
```

```
   (when (>= (|firstNonBlankPosition| line) 0) (push line acc))
   (pop lines))
  (nreverse acc)))
```

———-

12.8.3 defun checkSkipToken

[checkSkipIdentifierToken p474]
[checkSkipOpToken p474]

— defun checkSkipToken —

```
(defun |checkSkipToken| (u i m)
 (let ((str (string-trim '(#\space) u))) ; ignore leading spaces
  (if (alpha-char-p (elt str i))
  (|checkSkipIdentifierToken| str i m)
  (|checkSkipOpToken| str i m))))
```

———-

12.8.4 defun checkSplit2Words

[checkSplitBrace p474]

— defun checkSplit2Words —

```
(defun |checkSplit2Words| (u)
 (let (x verbatim z acc)
  (setq acc nil)
  (loop while u
   do
    (setq x (car u))
    (setq acc
     (cond
      ((string= x "\\end{verbatim}")
       (setq verbatim nil)
       (cons x acc))
      (verbatim (cons x acc))
      ((string= x "\\begin{verbatim}")
        (setq verbatim t)
        (cons x acc))
      ((setq z (|checkSplitBrace| x))
        (append (nreverse z) acc))
      (t (cons x acc))))
    (pop u))
  (nreverse acc)))
```

———-

12.9 Comment Checking Layer 8

12.9.1 defun checkAddIndented

[firstNonBlankPosition p485]
[strconc p??]
[checkAddSpaceSegments p478]
TPDHERE: Note that this function was missing without error, so may be junk

— defun checkAddIndented —

```
(defun |checkAddIndented| (x margin)
 (let (k)
  (setq k (|firstNonBlankPosition| x))
  (cond
    ((eql k -1) "\\blankline ")
    ((eql margin k) x)
    (t
     (strconc "\\indented{" (princ-to-string (- k margin)) "}{"
      (|checkAddSpaceSegments| (substring x k nil) 0) "}")))))
```

———-

12.9.2 defun checkDocMessage

[getdatabase p??]
[whoOwns p488]
[concat p??]
[$x p??]
[$constructorName p??]

— defun checkDocMessage —

```
(defun |checkDocMessage| (u)
 (let (sourcefile person middle)
 (declare (special |$constructorName| |$x|))
  (setq sourcefile (getdatabase |$constructorName| 'sourcefile))
  (setq person (or (|whoOwns| |$constructorName|) "---"))
  (setq middle
   (if (boundp '|$x|)
    (list "(" |$x| "): ")
    (list ": ")))
  (|concat| person ">" sourcefile "-->" |$constructorName| middle u)))
```

———-

12.9.3 defun checkExtract

[firstNonBlankPosition p485]
[substring? p??]
[charPosition p??]

[length p??]

— defun checkExtract —

```
(defun |checkExtract| (header lines)
 (let (line u margin firstLines m k j i acc)
  (loop while lines
   do
    (setq line (car lines))
    (setq k (|firstNonBlankPosition| line)) ; gives margin of description
    (if (|substring?| header line k)
      (return nil)
      (setq lines (cdr lines))))
  (cond
   ((null lines) nil)
   (t
    (setq u (car lines))
    (setq j (|charPosition| #\: u k))
    (setq margin k)
    (setq firstLines
     (if (not (eql (setq k (|firstNonBlankPosition| u (1+ j))) -1))
       (cons (substring u (1+ j) nil) (cdr lines))
       (cdr lines)))
    ; now look for another header; if found skip all rest of these lines
    (setq acc nil)
    (loop for line in firstLines
     do
      (setq m (|#| line))
      (cond
       ((eql (setq k (|firstNonBlankPosition| line)) -1) '|skip|)
       ((> k margin) '|skip|)
       ((null (upper-case-p (elt line k))) '|skip|)
       ((equal (setq j (|charPosition| #\: line k)) m) '|skip|)
       ((> j (setq i (|charPosition| #\space line (1+ k)))) '|skip|)
       (t (return nil)))
      (setq acc (cons line acc)))
    (nreverse acc)))))
```

———-

12.9.4 defun checkGetArgs

[maxindex p??]
[firstNonBlankPosition p485]
[checkGetArgs p470]
[stringPrefix? p??]
[getMatchingRightPren p485]
[charPosition p??]
[trimString p??]

— defun checkGetArgs —

```
(defun |checkGetArgs| (u)
```

```
 (let (m k acc i)
  (cond
   ((null (stringp u)) nil)
   (t
    (setq m (maxindex u))
    (setq k (|firstNonBlankPosition| u))
    (cond
     ((> k 0)
      (|checkGetArgs| (substring u k nil)))
     ((|stringPrefix?| "\\spad{" u)
      (setq k (or (|getMatchingRightPren| u 6 #\{ #\}) m))
      (|checkGetArgs| (substring u 6 (- k 6))))
     ((> (setq i (|charPosition| #\( u 0)) m)
      nil)
     ((not (eql (elt u m) #\)))
      nil)
     (t
      (do ()
          ((null (> m (setq k (|charPosition| #\, u (1+ i))))) nil)
        (setq acc
         (cons (|trimString| (substring u (1+ i) (1- (- k i)))) acc))
        (setq i k))
      (nreverse (cons (substring u (1+ i) (1- (- m i))) acc))))))))
```

———-

12.9.5 defun checkGetMargin

[firstNonBlankPosition p485]

— defun checkGetMargin —

```
(defun |checkGetMargin| (lines)
 (let (x k margin)
  (loop while lines
   do
    (setq x (car lines))
    (setq k (|firstNonBlankPosition| x))
    (unless (= k -1) (setq margin (if margin (min margin k) k)))
    (pop lines))
  (or margin 0)))
```

———-

12.9.6 defun checkGetParse

[ncParseFromString p**??**]
[removeBackslashes p487]

— defun checkGetParse —

```
(defun |checkGetParse| (s)
```

```
 (|ncParseFromString| (|removeBackslashes| s)))
```

———-

12.9.7 defun checkGetStringBeforeRightBrace

— defun checkGetStringBeforeRightBrace —

```
(defun |checkGetStringBeforeRightBrace| (u)
 (prog (x acc)
  (return
   (loop while u
    do
     (setq x (car u))
     (cond
      ((equal x #\})
       (let ((result ""))
        (loop for item in acc
         do (setq result (concatenate 'string item result)))
        (return result)))
      (t
       (setq acc (cons x acc))
       (setq u (cdr u))))))))
```

———-

12.9.8 defun checkIeEg

[checkIeEgfun p479]
[nreverse p??]

— defun checkIeEg —

```
(defun |checkIeEg| (u)
 (let (x verbatim z acc)
  (setq acc nil)
  (setq verbatim nil)
  (loop while u
   do
    (setq x (car u))
    (setq acc
     (cond
      ((equal x "\\end{verbatim}")
       (setq verbatim nil)
       (cons x acc))
      (verbatim (cons x acc))
      ((equal x "\\begin{verbatim}")
       (setq verbatim t)
       (cons x acc))
      ((setq z (|checkIeEgfun| x))
       (append (nreverse z) acc))
```

```
      (t (cons x acc))))
    (setq u (cdr u)))
   (nreverse acc)))
```

———-

12.9.9 defun checkIndentedLines

[firstNonBlankPosition p485]
[strconc p??]
[checkAddSpaceSegments p478]
[$charFauxNewline p??]

— defun checkIndentedLines —

```
(defun |checkIndentedLines| (u margin)
 (let (k s verbatim u2)
 (declare (special |$charFauxNewline|))
  (loop for x in u
   do
    (setq k (|firstNonBlankPosition| x))
    (cond
     ((eql k -1)
      (if verbatim
       (setq u2 (append u2 (list |$charFauxNewline|)))
       (setq u2 (append u2 (list "\\blankline ")))))
     (t
      (setq s (substring x k nil))
      (cond
       ((string= s "\\begin{verbatim}")
         (setq verbatim t)
         (setq u2 (append u2 (list s))))
       ((string= s "\\end{verbatim}")
         (setq verbatim nil)
         (setq u2 (append u2 (list s))))
       (verbatim
         (setq u2 (append u2 (list (substring x margin nil)))))
       ((eql margin k)
         (setq u2 (append u2 (list s))))
       (t
         (setq u2
          (append u2
           (list (strconc "\\indented{" (princ-to-string (- k margin))
                          "}{" (|checkAddSpaceSegments| s 0) "}"))))))))
  u2))
```

———-

12.9.10 defun checkSkipIdentifierToken

[checkAlphabetic p479]

— defun checkSkipIdentifierToken —

```
(defun |checkSkipIdentifierToken| (u i m)
 (do ()
   ((null (and (> m i) (|checkAlphabetic| (elt u i)))) nil)
  (setq i (1+ i)))
  (unless (= i m) i))
```

———-

12.9.11 defun checkSkipOpToken

[checkAlphabetic p479]
[member p??]
[$charDelimiters p??]

— defun checkSkipOpToken —

```
(defun |checkSkipOpToken| (u i m)
 (declare (special |$charDelimiters|))
  (do ()
      ((null (and (> m i)
                  (null (|checkAlphabetic| (elt u i)))
                  (null (|member| (elt u i) |$charDelimiters|))))
         nil)
   (setq i (1+ i)))
   (unless (= i m) i))
```

———-

12.9.12 defun checkSplitBrace

[charp p??]
[length p??]
[checkSplitBackslash p482]
[checkSplitBrace p474]
[checkSplitOn p483]
[checkSplitPunctuation p484]

— defun checkSplitBrace —

```
(defun |checkSplitBrace| (x)
 (let (m u)
  (cond
   ((charp x) (list x))
   ((eql (|#| x) 1) (list (elt x 0)))
   ((and (setq u (|checkSplitBackslash| x)) (cdr u))
     (let (result)
```

```
      (loop for y in u do (append result (|checkSplitBrace| y)))
      result))
    (t
      (setq m (maxindex x))
      (cond
        ((and (setq u (|checkSplitOn| x)) (cdr u))
         (let (result)
          (loop for y in u do (append result (|checkSplitBrace| y)))
          result))
        ((and (setq u (|checkSplitPunctuation| x)) (cdr u))
         (let (result)
          (loop for y in u do (append result (|checkSplitBrace| y)))
          result))
        (t (list x)))))))
```

——————

12.9.13 defun checkTrimCommented

[length p??]
[htcharPosition p486]

— defun checkTrimCommented —

```
(defun |checkTrimCommented| (line)
 (let (n k)
  (setq n (|#| line))
  (setq k (|htcharPosition| #\% line 0))
  (cond
   ((eql k 0) "")
   ((or (>= k (1- n)) (not (eql (elt line (1+ k)) #\%))) line)
   ((> (|#| line) k) (substring line 0 k))
   (t line))))
```

——————

12.9.14 defun newString2Words

[newWordFrom p487]
[nreverse0 p??]

— defun newString2Words —

```
(defun |newString2Words| (z)
 (let (m tmp1 w i result)
  (cond
   ((null (stringp z)) (list z))
   (t
    (setq m (maxindex z))
    (cond
     ((eql m -1) nil)
     (t
```

```
      (setq i 0)
      (do () ; [w while newWordFrom(l,i,m) is [w,i]]
          ((null (progn
                  (setq tmp1 (|newWordFrom| z i m))
                  (and (consp tmp1)
                       (progn
                        (setq w (qcar tmp1))
                        (and (consp (qcdr tmp1))
                             (eq (qcddr tmp1) nil)
                             (progn
                              (setq i (qcadr tmp1))
                              t))))))
             (nreverse0 result))
          (setq result (cons (qcar tmp1) result))))))))
```

———–

12.10 Comment Checking Layer 9

12.10.1 defun checkAddBackSlashes

[strconc p??]
[maxindex p??]
[checkAddBackSlashes p476]
[$charEscapeList p??]

— defun checkAddBackSlashes —

```
(defun |checkAddBackSlashes| (s)
 (let (c m char insertIndex k)
 (declare (special |$charEscapeList|))
  (cond
   ((or (and (charp s) (setq c s))
        (and (eql (|#| s) 1) (setq c (elt s 0))))
     (if (member s |$charEscapeList|)
      (strconc #\\ c)
      s))
    (t
     (setq k 0)
     (setq m (maxindex s))
     (setq insertIndex nil)
     (loop while (< k m)
      do
       (setq char (elt s k))
       (cond
        ((char= char #\\) (setq k (+ k 2)))
        ((member char |$charEscapeList|) (return (setq insertIndex k))))
        (setq k (1+ k)))
     (cond
      (insertIndex
       (|checkAddBackSlashes|
        (strconc (substring s 0 insertIndex) #\\ (elt s k)
```

```
            (substring s (1+ insertIndex) nil))))
     (T s))))))
```

———-

12.10.2 defun checkAddMacros

[lassoc p??]
[nreverse p??]
[$HTmacs p??]

— defun checkAddMacros —

```
(defun |checkAddMacros| (u)
 (let (x verbatim y acc)
 (declare (special |$HTmacs|))
  (loop while u
   do
   (setq x (car u))
   (setq acc
    (cond
     ((string= x "\\end{verbatim}")
       (setq verbatim nil)
       (cons x acc))
     (verbatim
       (cons x acc))
     ((string= x "\\begin{verbatim}")
       (setq verbatim t)
       (cons x acc))
     ((setq y (lassoc x |$HTmacs|))
       (append y acc))
     (t (cons x acc))))
   (pop u))
  (nreverse acc)))
```

———-

12.10.3 defun checkAddPeriod

[setelt p??]
[maxindex p??]

— defun checkAddPeriod —

```
(defun |checkAddPeriod| (s)
 (let (m lastChar)
  (setq m (maxindex s))
  (setq lastChar (elt s m))
  (cond
   ((or (char= lastChar #\!) (char= lastChar #\?) (char= lastChar #\.)) s)
   ((or (char= lastChar #\,) (char= lastChar #\;))
     (setelt s m #\.)
```

```
     s)
   (t s))))
```

———-

12.10.4 defun checkAddSpaceSegments

[checkAddSpaceSegments p478]
[maxindex p??]
[charPosition p??]
[strconc p??]
[$charBlank p??]

— defun checkAddSpaceSegments —

```
(defun |checkAddSpaceSegments| (u k)
 (let (m i j n)
  (setq m (maxindex u))
  (setq i (|charPosition| #\space u k))
  (cond
   ((> i m) u)
   (t
    (setq j i)
    (loop while (and (incf j) (char= (elt u j) #\space)))
    (setq n (- j i)) ; number of blanks
    (if (> n 1)
     (strconc (substring u 0 i) "\\space{" (princ-to-string n) "}"
               (|checkAddSpaceSegments| (substring u (+ i n) nil) 0))
     (|checkAddSpaceSegments| u j))))))
```

———-

12.10.5 defun checkAddSpaces

[$charBlank p??]
[$charFauxNewline p??]

— defun checkAddSpaces —

```
(defun |checkAddSpaces| (u)
 (let (u2 space i)
 (declare (special |$charFauxNewline|))
  (cond
   ((null u) nil)
   ((null (cdr u)) u)
   (t
    (setq space #\space)
    (setq i 0)
    (loop for f in u
     do
      (incf i)
```

```
      (when (string= f "\\begin{verbatim}")
        (setq space |$charFauxNewline|)
        (unless u2 (setq u2 (list space))))
      (if (> i 1)
        (setq u2 (append u2 (list space f)))
        (setq u2 (append u2 (list f))))
      (when (string= f "\\end{verbatim}")
        (setq u2 (append u2 (list space)))
        (setq space #\space)))
    u2))))
```

———-

12.10.6 defun checkAlphabetic

[$charIdentifierEndings p??]

— defun checkAlphabetic —

```
(defun |checkAlphabetic| (c)
 (declare (special |$charIdentifierEndings|))
 (or (alpha-char-p c) (digitp c) (member c |$charIdentifierEndings|)))
```

———-

12.10.7 defun checkIeEgfun

[maxindex p??]
[checkIeEgFun p??]

— defun checkIeEgfun —

```
(defun |checkIeEgfun| (x)
 (let (m key firstPart result)
  (cond
   ((characterp x) nil)
   ((equal x "") nil)
   (t
    (setq m (maxindex x))
    (loop for k from 0 to (- m 3)
     do
      (cond
       ((and
          (equal (elt x (1+ k)) #\.)
          (equal (elt x (+ k 3)) #\.)
          (or
           (and
            (equal (elt x k) #\i)
            (equal (elt x (+ k 2)) #\e)
            (setq key "that is"))
           (and
            (equal (elt x k) #\e)
```

```
            (equal (elt x (+ k 2)) #\g)
            (setq key "for example"))))
         (progn
          (setq firstPart (when (> k 0) (cons (substring x 0 k) nil)))
          (setq result
           (append firstPart
            (cons "\\spadignore{"
             (cons (substring x k 4)
              (cons "}"
               (|checkIeEgfun| (substring x (+ k 4) nil))))))))))
    result))))
```

———-

12.10.8 defun checkIsValidType

This function returns ok if correct, form is wrong number of arguments, nil if unknown
[length p??]
[checkIsValidType p480]
[constructor? p??]
[abbreviation? p??]
[getdatabase p??]

— defun checkIsValidType —

```
(defun |checkIsValidType| (form)
 (labels (
  (fn (form coSig)
   (cond
    ((not (eql (|#| form) (|#| coSig))) form)
    ((let (result)
      (loop for x in (rest form)
            for flag in (rest coSig)
       do (when flag (setq result (or result (null (|checkIsValidType| x))))))
      result)
     nil)
    (t '|ok|))))
 (let (op args conname)
  (cond
   ((atom form) '|ok|)
   (t
    (setq op (car form))
    (setq args (cdr form))
    (setq conname
     (if (|constructor?| op)
      op
      (|abbreviation?| op)))
    (when conname (fn form (getdatabase conname 'cosig))))))))
```

———-

12.10.9 defun checkLookForLeftBrace

[$charBlank p??]

— defun checkLookForLeftBrace —

```
(defun |checkLookForLeftBrace| (u)
  (loop while u
   do
    (cond
     ((equal (car u) #\{) (return (car u)))
     ((not (eql (car u) #\space)) (return nil))
     (t (pop u))))
  u)
```

———-

12.10.10 defun checkLookForRightBrace

This returns a line beginning with right brace

— defun checkLookForRightBrace —

```
(defun |checkLookForRightBrace| (u)
 (let (found count)
  (setq count 0)
  (loop while u
   do
   (cond
    ((equal (car u) #\})
     (if (eql count 0)
      (return (setq found u))
      (setq count (1- count))))
    ((equal (car u) #\{)
      (setq count (1+ count))))
    (pop u))
  found))
```

———-

12.10.11 defun checkNumOfArgs

A nil return implies that the argument list length does not match [opOf p??]
[constructor? p??]
[abbreviation? p??]
[getdatabase p??]

— defun checkNumOfArgs —

```
(defun |checkNumOfArgs| (conform)
 (let (conname)
  (setq conname (|opOf| conform))
  (when (or (|constructor?| conname) (setq conname (|abbreviation?| conname)))
```

```
    (|#| (getdatabase conname 'constructorargs)))))
```

———-

12.10.12 defun checkSayBracket

— defun checkSayBracket —

```
(defun |checkSayBracket| (x)
 (cond
  ((or (char= x #\() (char= x #\))) "pren")
  ((or (char= x #\{) (char= x #\})) "brace")
  (t "bracket")))
```

———-

12.10.13 defun checkSkipBlanks

[$charBlank p??]

— defun checkSkipBlanks —

```
(defun |checkSkipBlanks| (u i m)
  (do ()
      ((null (and (> m i) (equal (elt u i) #\space))) nil)
   (setq i (1+ i)))
  (unless (= i m) i))
```

———-

12.10.14 defun checkSplitBackslash

[checkSplitBackslash p482]
[maxindex p??]
[charPosition p??]

— defun checkSplitBackslash —

```
(defun |checkSplitBackslash| (x)
 (let (m k u v)
  (cond
   ((null (stringp x)) (list x))
   (t
    (setq m (maxindex x))
    (cond
     ((> m (setq k (|charPosition| #\\ x 0)))
      (cond
       ((or (eql m 1) (alpha-char-p (elt x (1+ k)))) ;starts with backslash so
        (if (> m (setq k (|charPosition| #\\ x 1)))
         ; yes, another backslash
```

```
        (cons (substring x 0 k) (|checkSplitBackslash| (substring x k nil)))
        ; no, just return the line
        (list x)))
      ((eql k 0)
        ; starts with backspace but x.1 is not a letter; break it up
        (cons (substring x 0 2)
          (|checkSplitBackslash| (substring x 2 nil))))
      (t
        (setq u (substring x 0 k))
        (setq v (substring x k 2))
        (if (= (1+ k) m)
           (list u v)
          (cons u
           (cons v
            (|checkSplitBackslash|
             (substring x (+ k 2) nil))))))))
    (t (list x)))))))
```

———

12.10.15 defun checkSplitOn

[checkSplitOn p483]
[charp p??]
[maxindex p??]
[charPosition p??]
[$charSplitList p??]

— defun checkSplitOn —

```
(defun |checkSplitOn| (x)
 (let (m char k z)
 (declare (special |$charSplitList|))
  (cond
   ((charp x) (list x))
   (t
    (setq z |$charSplitList|)
    (setq m (maxindex x))
    (loop while z
     do
      (setq char (car z))
      (cond
       ((and (eql m 0) (equal (elt x 0) char))
         (return (setq k -1)))
       (t
         (setq k (|charPosition| char x 0))
         (cond
          ((and (> k 0) (equal (elt x (1- k)) #\\)) (list x))
          ((<= k m) (return k)))))
      (pop z))
    (cond
     ((null z) (list x))
     ((eql k -1) (list char))
```

```
    ((eql k 0) (list char (substring x 1 nil)))
    ((eql k (maxindex x)) (list (substring x 0 k) char))
    (t
      (cons (substring x 0 k)
            (cons char (|checkSplitOn| (substring x (1+ k) nil))))))))))
```

———-

12.10.16 defun checkSplitPunctuation

[charp p??]
[maxindex p??]
[checkSplitPunctuation p484]
[charPosition p??]
[hget p??]
[$htMacroTable p??]

— defun checkSplitPunctuation —

```
(defun |checkSplitPunctuation| (x)
 (let (m lastchar v k u)
 (declare (special |$htMacroTable|))
  (cond
   ((charp x) (list x))
   (t
     (setq m (maxindex x))
     (cond
      ((> 1 m) (list x))
      (t
       (setq lastchar (elt x m))
       (cond
        ((and (equal lastchar #\.)
              (equal (elt x (1- m)) #\.))
          (cond
           ((eql m 1) (list x))
           ((and (> m 3) (equal (elt x (- m 2)) #\.))
             (append (|checkSplitPunctuation| (substring x 0 (- m 2)))
                     (list "...")))
           (t
             (append (|checkSplitPunctuation| (substring x 0 (1- m)))
                     (list "..")))))
        ((or (equal lastchar #\.)
             (equal lastchar #\;)
             (equal lastchar #\,))
          (list (substring x 0 m) lastchar))
        ((and (> m 1) (equal (elt x (1- m)) #\'))
          (list (substring x 0 (1- m)) (substring x (1- m) nil)))
        ((> m (setq k (|charPosition| #\\ x 0)))
          (cond
           ((eql k 0)
            (cond
             ((or (eql m 1) (hget |$htMacroTable| x) (alpha-char-p (elt x 1)))
```

```
          (list x))
         (t
          (setq v (substring x 2 nil))
           (cons (substring x 0 2) (|checkSplitPunctuation| v)))))
        (t
         (setq u (substring x 0 k))
         (setq v (substring x k nil))
         (append (|checkSplitPunctuation| u)
                 (|checkSplitPunctuation| v)))))
      ((> m (setq k (|charPosition| #\- x 1)))
         (setq u (substring x (1+ k) nil))
         (cons (substring x 0 k)
           (cons #\- (|checkSplitPunctuation| u))))
      (t
       (list x))))))))
```

———-

12.10.17 defun firstNonBlankPosition

[maxindex p??]

— defun firstNonBlankPosition —

```
(defun |firstNonBlankPosition| (&rest therest)
 (let ((x (car therest)) (options (cdr therest)) start k)
  (setq start (or (ifcar options) 0))
  (setq k -1)
  (loop for i from start to (maxindex x)
   do (when (not (eql (elt x i) #\space)) (return (setq k i))))
  k))
```

———-

12.10.18 defun getMatchingRightPren

[maxindex p??]

— defun getMatchingRightPren —

```
(defun |getMatchingRightPren| (u j open close)
 (let (m c found count)
  (setq count 0)
  (setq m (maxindex u))
  (loop for i from j to m
   do
    (setq c (elt u i))
    (cond
     ((equal c close)
      (if (eql count 0)
       (return (setq found i))
       (setq count (1- count))))
```

```
    ((equal c open)
      (setq count (1+ count)))))
  found))
```

———-

12.10.19 defun hasNoVowels

[maxindex p??]

— defun hasNoVowels —

```
(defun |hasNoVowels| (x)
 (labels (
  (isVowel (c)
   (or (eq c #\a) (eq c #\e) (eq c #\i) (eq c #\o) (eq c #\u)
       (eq c #\A) (eq c #\E) (eq c #\I) (eq c #\O) (eq c #\U))))
 (let (max)
  (setq max (maxindex x))
  (cond
   ((char= (elt x max) #\y) nil)
   (t
    (let ((result t))
     (loop for i from 0 to max
      do (setq result (and result (null (isVowel (elt x i))))))
     result))))))
```

———-

12.10.20 defun htcharPosition

[length p??]
[charPosition p??]
[htcharPosition p486]

— defun htcharPosition —

```
(defun |htcharPosition| (char line i)
 (let (m k)
  (setq m (|#| line))
  (setq k (|charPosition| char line i))
  (cond
   ((eql k m) k)
   ((> k 0)
    (if (not (eql (elt line (1- k)) #\\))
     k
     (|htcharPosition| char line (1+ k))))
   (t 0))))
```

———-

12.10.21 defun newWordFrom

[$stringFauxNewline p??]
[$charBlank p??]
[$charFauxNewline p??]

— defun newWordFrom —

```
(defun |newWordFrom| (z i m)
 (let (ch done buf)
 (declare (special |$charFauxNewline| |$stringFauxNewline|))
  (loop while (and (<= i m) (char= (elt z i) #\space)) do (incf i))
  (cond
   ((> i m) nil)
   (t
    (setq buf "")
    (setq ch (elt z i))
    (cond
     ((equal ch |$charFauxNewline|)
       (list |$stringFauxNewline| (1+ i)))
     (t
      (setq done nil)
      (loop while (and (<= i m) (null done))
       do
        (setq ch (elt z i))
        (cond
         ((or (equal ch #\space) (equal ch |$charFauxNewline|))
           (setq done t))
         (t
          (setq buf (strconc buf ch))
          (setq i (1+ i)))))
      (list buf i)))))))
```

———-

12.10.22 defun removeBackslashes

[charPosition p??]
[removeBackslashes p487]
[strconc p??]
[length p??]

— defun removeBackslashes —

```
(defun |removeBackslashes| (s)
 (let (k)
  (cond
   ((string= s "") "")
   ((> (|#| s) (setq k (|charPosition| #\\ s 0)))
    (if (eql k 0)
     (|removeBackslashes| (substring s 1 nil))
     (strconc (substring s 0 k)
              (|removeBackslashes| (substring s (1+ k) nil)))))
```

```
   (t s))))
```

———-

12.10.23 defun whoOwns

This function always returns nil in the current system. Since it has no side effects we define it to return nil. [getdatabase p??]
[strconc p??]
[awk p??]
[shut p??]
[$exposeFlag p??]

— **defun whoOwns** —

```
(defun |whoOwns| (con)
 (declare (ignore con))
 nil)
; (let (filename quoteChar instream value)
; (declare (special |$exposeFlag|))
;  (cond
;   ((null |$exposeFlag|) nil)
;   (t
;    (setq filename (getdatabase con 'sourcefile))
;    (setq quoteChar #\")
;    (obey (strconc "awk '$2 == " quoteChar filename quoteChar
;                   " {print $1}' whofiles > /tmp/temp"))
;    (setq instream (make-instream "/tmp/temp"))
;    (setq value (unless (eofp instream) (readline instream)))
;    (shut instream)
;  value))))
```

———-

Chapter 13

Utility Functions

13.0.24 defun translabel

[translabel1 p489]

— defun translabel —

```
(defun translabel (x al)
 (translabel1 x al) x)
```

———-

13.0.25 defun translabel1

[refvecp p??]
[maxindex p??]
[translabel1 p489]
[lassoc p??]

— defun translabel1 —

```
(defun translabel1 (x al)
 "Transforms X according to AL = ((<label> . Sexpr) ..)."
  (cond
   ((refvecp x)
    (do ((i 0 (1+ i)) (k (maxindex x)))
        ((> i k))
      (if (let ((y (lassoc (elt x i) al))) (setelt x i y))
       (translabel1 (elt x i) al))))
   ((atom x) nil)
   ((let ((y (lassoc (first x) al)))
          (if y (setf (first x) y) (translabel1 (cdr x) al))))
   ((translabel1 (first x) al) (translabel1 (cdr x) al))))
```

———-

13.0.26 defun displayPreCompilationErrors

[length p??]
[remdup p??]
[sayBrightly p??]
[sayMath p??]
[$postStack p??]
[$topOp p??]
[$InteractiveMode p??]

— defun displayPreCompilationErrors —

```
(defun |displayPreCompilationErrors| ()
 (let (n errors heading)
  (declare (special |$postStack| |$topOp| |$InteractiveMode|))
  (setq n (|#| (setq |$postStack| (remdup (nreverse |$postStack|)))))
  (unless (eql n 0)
    (setq errors (cond ((> n 1) "errors") (t "error")))
    (cond
     (|$InteractiveMode|
      (|sayBrightly| (list "   Semantic " errors " detected: ")))
     (t
      (setq heading
       (if (not (eq |$topOp| '|$topOp|))
        (list "   " |$topOp| " has")
        (list "   You have")))
      (|sayBrightly|
       (append heading (list n "precompilation " errors ":" )))))
    (cond
     ((> n 1)
       (let ((i 1))
        (dolist (x |$postStack|)
          (|sayMath| (cons "   " (cons i (cons ") " x)))))))
     (t (|sayMath| (cons "    " (car |$postStack|)))))
    (terpri))))
```

———-

13.0.27 defun bumperrorcount

[$InteractiveMode p??]
[$spad-errors p??]

— defun bumperrorcount —

```
(defun bumperrorcount (kind)
 (declare (special |$InteractiveMode| $spad_errors))
 (unless |$InteractiveMode|
  (let ((index (case kind
                (|syntax| 0)
                (|precompilation| 1)
                (|semantic| 2)
                (t (error (break "BUMPERRORCOUNT: kind=~s~%" kind))))))
```

```
    (setelt $spad_errors index (1+ (elt $spad_errors index))))))
```

———-

13.0.28 defun parseTranCheckForRecord

[postError p341]
[parseTran p99]

— defun parseTranCheckForRecord —

```
(defun |parseTranCheckForRecord| (x op)
 (declare (ignore op))
 (let (tmp3)
  (setq x (|parseTran| x))
  (cond
   ((and (consp x) (eq (qfirst x) '|Record|))
     (cond
      ((do ((z nil tmp3) (tmp4 (qrest x) (cdr tmp4)) (y nil))
           ((or z (atom tmp4)) tmp3)
          (setq y (car tmp4))
          (cond
           ((null (and (consp y) (eq (qfirst y) '|:|) (consp (qrest y))
                  (consp (qcddr y)) (eq (qcdddr y) nil)))
              (setq tmp3 (or tmp3 y)))))
          (postError (list "   Constructor" x "has missing label" )))
       (t x)))
   (t x))))
```

———-

13.0.29 defun makeSimplePredicateOrNil

[isSimple p??]
[isAlmostSimple p??]
[wrapSEQExit p??]

— defun makeSimplePredicateOrNil —

```
(defun |makeSimplePredicateOrNil| (p)
 (let (u g)
  (cond
   ((|isSimple| p) nil)
   ((setq u (|isAlmostSimple| p)) u)
   (t (|wrapSEQExit| (list (list 'let (setq g (gensym)) p) g))))))
```

———-

13.0.30 defun parse-spadstring

[match-current-token p413]
[token-symbol p??]
[push-reduction p421]
[advance-token p415]

— defun parse-spadstring —

```
(defun parse-spadstring ()
  (let* ((tok (match-current-token 'spadstring))
         (symbol (if tok (token-symbol tok))))
    (when tok
      (push-reduction 'spadstring-token (copy-tree symbol))
      (advance-token)
      t)))
```

———-

13.0.31 defun parse-string

[match-current-token p413]
[token-symbol p??]
[push-reduction p421]
[advance-token p415]

— defun parse-string —

```
(defun parse-string ()
  (let* ((tok (match-current-token 'string))
         (symbol (if tok (token-symbol tok))))
    (when tok
      (push-reduction 'string-token (copy-tree symbol))
      (advance-token)
      t)))
```

———-

13.0.32 defun parse-identifier

[match-current-token p413]
[token-symbol p??]
[push-reduction p421]
[advance-token p415]

— defun parse-identifier —

```
(defun parse-identifier ()
  (let* ((tok (match-current-token 'identifier))
         (symbol (if tok (token-symbol tok))))
    (when tok
```

```
      (push-reduction 'identifier-token (copy-tree symbol))
      (advance-token)
      t)))
```

———-

13.0.33 defun parse-number

[match-current-token p413]
[token-symbol p??]
[push-reduction p421]
[advance-token p415]

— **defun parse-number** —

```
(defun parse-number ()
  (let* ((tok (match-current-token 'number))
         (symbol (if tok (token-symbol tok))))
    (when tok
      (push-reduction 'number-token (copy-tree symbol))
      (advance-token)
      t)))
```

———-

13.0.34 defun parse-keyword

[match-current-token p413]
[token-symbol p??]
[push-reduction p421]
[advance-token p415]

— **defun parse-keyword** —

```
(defun parse-keyword ()
  (let* ((tok (match-current-token 'keyword))
         (symbol (if tok (token-symbol tok))))
    (when tok
      (push-reduction 'keyword-token (copy-tree symbol))
      (advance-token)
      t)))
```

———-

13.0.35 defun parse-argument-designator

[push-reduction p421]
[match-current-token p413]
[token-symbol p??]
[advance-token p415]

— defun parse-argument-designator —

```
(defun parse-argument-designator ()
  (let* ((tok (match-current-token 'argument-designator))
         (symbol (if tok (token-symbol tok))))
    (when tok
      (push-reduction 'argument-designator-token (copy-tree symbol))
      (advance-token)
      t)))
```

———-

13.0.36 defun checkWarning

[postError p341]
[concat p??]

— defun checkWarning —

```
(defun |checkWarning| (msg)
 (postError (|concat| "Parsing error: " msg)))
```

———-

13.0.37 defun tuple2List

[tuple2List p494]
[postTranSegment p354]
[postTran p338]
[$boot p??]
[$InteractiveMode p??]

— defun tuple2List —

```
(defun |tuple2List| (arg)
 (let (u p q)
 (declare (special |$InteractiveMode| $boot))
  (when (consp arg)
   (setq u (|tuple2List| (qrest arg)))
   (cond
    ((and (consp (qfirst arg)) (eq (qcaar arg) 'segment)
          (consp (qcdar arg))
          (consp (qcddar arg))
          (eq (qcdddar arg) nil))
       (setq p (qcadar arg))
       (setq q (qcaddar arg))
       (cond
        ((null u) (list '|construct| (|postTranSegment| p q)))
        ((and |$InteractiveMode| (null $boot))
          (cons '|append|
           (cons (list '|construct| (|postTranSegment| p q))
```

```
              (list (|tuple2List| (qrest arg))))))
       (t
        (cons '|nconc|
         (cons (list '|construct| (|postTranSegment| p q))
              (list (|tuple2List| (qrest arg))))))))
   ((null u) (list '|construct|  (|postTran| (qfirst arg))))
   (t (list '|cons| (|postTran| (qfirst arg)) (|tuple2List| (qrest arg)))))))
```

———-

13.0.38 defmacro pop-stack-1

[reduction-value p??]
[Pop-Reduction p496]

— defmacro pop-stack-1 —

```
(defmacro pop-stack-1 () '(reduction-value (Pop-Reduction)))
```

———-

13.0.39 defmacro pop-stack-2

[stack-push p94]
[reduction-value p??]
[Pop-Reduction p496]

— defmacro pop-stack-2 —

```
(defmacro pop-stack-2 ()
  `(let* ((top (Pop-Reduction)) (next (Pop-Reduction)))
     (stack-push top Reduce-Stack)
     (reduction-value next)))
```

———-

13.0.40 defmacro pop-stack-3

[stack-push p94]
[reduction-value p??]
[Pop-Reduction p496]

— defmacro pop-stack-3 —

```
(defmacro pop-stack-3 ()
  `(let* ((top (Pop-Reduction)) (next (Pop-Reduction)) (nnext (Pop-Reduction)))
     (stack-push next Reduce-Stack)
     (stack-push top Reduce-Stack)
     (reduction-value nnext)))
```

———-

13.0.41 defmacro pop-stack-4

[stack-push p94]
[reduction-value p??]
[Pop-Reduction p496]

— defmacro pop-stack-4 —

```
(defmacro pop-stack-4 ()
  `(let* ((top (Pop-Reduction))
          (next (Pop-Reduction))
          (nnext (Pop-Reduction))
          (nnnext (Pop-Reduction)))
     (stack-push nnext Reduce-Stack)
     (stack-push next Reduce-Stack)
     (stack-push top Reduce-Stack)
     (reduction-value nnnext)))
```

———-

13.0.42 defmacro nth-stack

[stack-store p??]
[reduction-value p??]

— defmacro nth-stack —

```
(defmacro nth-stack (x)
  `(reduction-value (nth (1- ,x) (stack-store Reduce-Stack))))
```

———-

13.0.43 defun Pop-Reduction

[stack-pop p94]

— defun Pop-Reduction —

```
(defun Pop-Reduction () (stack-pop Reduce-Stack))
```

———-

13.0.44 defun addclose

[suffix p??]

— defun addclose —

```
(defun addclose (line char)
 (cond
  ((char= (char line (maxindex line)) #\; )
    (setelt line (maxindex line) char)
    (if (char= char #\;) line (suffix #\; line)))
  ((suffix char line))))
```

———-

13.0.45 defun blankp

— defun blankp —

```
(defun blankp (char)
 (or (eq char #\Space) (eq char #\tab)))
```

———-

13.0.46 defun drop

Return a pointer to the Nth cons of X, counting 0 as the first cons. [drop p497]
[take p??]
[croak p??]

— defun drop —

```
(defun drop (n x &aux m)
 (cond
  ((eql n 0) x)
  ((> n 0) (drop (1- n) (cdr x)))
  ((>= (setq m (+ (length x) n)) 0) (take m x))
  ((croak (list "Bad args to DROP" n x)))))
```

———-

13.0.47 defun escaped

— defun escaped —

```
(defun escaped (str n)
 (and (> n 0) (eq (char str (1- n)) #\_)))
```

———-

13.0.48 defvar $comblocklist

— initvars —

```
(defvar $comblocklist nil "a dynamic lists of comments for this block")
```

———-

13.0.49 defun fincomblock

- NUM is the line number of the current line
- OLDNUMS is the list of line numbers of previous lines
- OLDLOCS is the list of previous indentation locations
- NCBLOCK is the current comment block

[preparse-echo p93]
[$comblocklist p498]
[$EchoLineStack p??]

— defun fincomblock —

```
(defun fincomblock (num oldnums oldlocs ncblock linelist)
 (declare (special $EchoLineStack $comblocklist))
 (push
  (cond
   ((eql (car ncblock) 0) (cons (1- num) (reverse (cdr ncblock))))
    ;; comment for constructor itself paired with 1st line -1
   (t
    (when $EchoLineStack
     (setq num (pop $EchoLineStack))
     (preparse-echo linelist)
     (setq $EchoLineStack (list num)))
    (cons            ;; scan backwards for line to left of current
     (do ((onums oldnums (cdr onums))
          (olocs oldlocs (cdr olocs))
          (sloc (car ncblock)))
         ((null onums) nil)
       (when (and (numberp (car olocs)) (<= (car olocs) sloc))
         (return (car onums))))
     (reverse (cdr ncblock)))))
   $comblocklist))
```

———-

13.0.50 defun indent-pos

[next-tab-loc p499]

— defun indent-pos —

```
(defun indent-pos (str)
  (do ((i 0 (1+ i)) (pos 0))
      ((>= i (length str)) nil)
   (case (char str i)
    (#\space (incf pos))
    (#\tab (setq pos (next-tab-loc pos)))
    (otherwise (return pos)))))
```

———-

13.0.51 defun infixtok

[string2id-n p??]

— defun infixtok —

```
(defun infixtok (s)
 (member (string2id-n s 1) '(|then| |else|) :test #'eq))
```

———-

13.0.52 defun is-console

[fp-output-stream p??]
[*terminal-io* p??]

is-console : **Stream** → **Boolean**
— defun is-console —

```
(defun is-console (stream)
  (and (streamp stream) (output-stream-p stream)
       (eq (system:fp-output-stream stream)
           (system:fp-output-stream *terminal-io*))))
```

———-

13.0.53 defun next-tab-loc

— defun next-tab-loc —

```
(defun next-tab-loc (i)
 (* (1+ (truncate i 8)) 8))
```

———-

13.0.54 defun nonblankloc

[blankp p497]

— defun nonblankloc —

```
(defun nonblankloc (str)
 (position-if-not #'blankp str))
```

———-

13.0.55 defun parseprint

— defun parseprint —

```
(defun parseprint (l)
 (when l
  (format t "~&~%       ***       PREPARSE      ***~%~%")
  (dolist (x l) (format t "~5d. ~a~%" (car x) (cdr x)))
  (format t "~%")))
```

———-

13.0.56 defun skip-to-endif

[skip-to-endif initial-substring (vol5)]
[preparseReadLine p??]
[preparseReadLine1 p91]
[skip-to-endif p500]

— defun skip-to-endif —

```
(defun skip-to-endif (x)
 (let (line ind tmp1)
  (setq tmp1 (preparseReadLine1))
  (setq ind (car tmp1))
  (setq line (cdr tmp1))
  (cond
   ((not (stringp line)) (cons ind line))
   ((initial-substring line ")endif") (preparseReadLine x))
   ((initial-substring line ")fin") (cons ind nil))
   (t (skip-to-endif x)))))
```

———-

Chapter 14

The Compiler

14.0.57 defvar $newConlist

A list of new constructors discovered during compile. These are used in a call to `extendLocalLibdb` when a user compiles new local code.

— **initvars** —

```
(defvar |$newConlist| nil
 "A list of new constructors discovered during compile ")
```

———-

14.1 Compiling EQ.spad

Given the top level command:

```
)co EQ
```

The default call chain looks like:

```
1> (|compiler| ...)
 2> (|compileSpad2Cmd| ...)
   Compiling AXIOM source code from file /tmp/A.spad
  3> (|compilerDoit| ...)
   4> (|/RQ,LIB|)
    5> (/RF-1 ...)
     6> (SPAD ...)
   AXSERV abbreviates package AxiomServer
      7> (S-PROCESS ...)
       8> (|compTopLevel| ...)
        9> (|compOrCroak| ...)
         10> (|compOrCroak1| ...)
          11> (|comp| ...)
           12> (|compNoStacking| ...)
            13> (|comp2| ...)
             14> (|comp3| ...)
              15> (|compExpression| ...)
```

```
*               16> (|compWhere| ...)
                 17> (|comp| ...)
                  18> (|compNoStacking| ...)
                   19> (|comp2| ...)
                    20> (|comp3| ...)
                     21> (|compExpression| ...)
                      22> (|compSeq| ...)
                       23> (|compSeq1| ...)
                        24> (|compSeqItem| ...)
                         25> (|comp| ...)
                          26> (|compNoStacking| ...)
                           27> (|comp2| ...)
                            28> (|comp3| ...)
                             29> (|compExpression| ...)
                             <29 (|compExpression| ...)
                            <28 (|comp3| ...)
                           <27 (|comp2| ...)
                          <26 (|compNoStacking| ...)
                         <25 (|comp| ...)
                        <24 (|compSeqItem| ...)
                        24> (|compSeqItem| ...)
                         25> (|comp| ...)
                          26> (|compNoStacking| ...)
                           27> (|comp2| ...)
                            28> (|comp3| ...)
                             29> (|compExpression| ...)
                              30> (|compExit| ...)
                               31> (|comp| ...)
                                32> (|compNoStacking| ...)
                                 33> (|comp2| ...)
                                  34> (|comp3| ...)
                                   35> (|compExpression| ...)
                                   <35 (|compExpression| ...)
                                  <34 (|comp3| ...)
                                 <33 (|comp2| ...)
                                <32 (|compNoStacking| ...)
                               <31 (|comp| ...)
                               31> (|modifyModeStack| ...)
                               <31 (|modifyModeStack| ...)
                              <30 (|compExit| ...)
                             <29 (|compExpression| ...)
                            <28 (|comp3| ...)
                           <27 (|comp2| ...)
                          <26 (|compNoStacking| ...)
                         <25 (|comp| ...)
                        <24 (|compSeqItem| ...)
                        24> (|replaceExitEtc| ...)
                         25> (|replaceExitEtc,fn| ...)
                          26> (|replaceExitEtc| ...)
                           27> (|replaceExitEtc,fn| ...)
                            28> (|replaceExitEtc| ...)
                             29> (|replaceExitEtc,fn| ...)
                             <29 (|replaceExitEtc,fn| ...)
                            <28 (|replaceExitEtc| ...)
```

```
           28> (|replaceExitEtc| ...)
            29> (|replaceExitEtc,fn| ...)
            <29 (|replaceExitEtc,fn| ...)
           <28 (|replaceExitEtc| ...)
          <27 (|replaceExitEtc,fn| ...)
         <26 (|replaceExitEtc| ...)
         26> (|replaceExitEtc| ...)
          27> (|replaceExitEtc,fn| ...)
           28> (|replaceExitEtc| ...)
            29> (|replaceExitEtc,fn| ...)
             30> (|replaceExitEtc| ...)
              31> (|replaceExitEtc,fn| ...)
               32> (|replaceExitEtc| ...)
                33> (|replaceExitEtc,fn| ...)
                <33 (|replaceExitEtc,fn| ...)
               <32 (|replaceExitEtc| ...)
               32> (|replaceExitEtc| ...)
                33> (|replaceExitEtc,fn| ...)
                <33 (|replaceExitEtc,fn| ...)
               <32 (|replaceExitEtc| ...)
              <31 (|replaceExitEtc,fn| ...)
             <30 (|replaceExitEtc| ...)
             30> (|convertOrCroak| ...)
              31> (|convert| ...)
              <31 (|convert| ...)
             <30 (|convertOrCroak| ...)
            <29 (|replaceExitEtc,fn| ...)
           <28 (|replaceExitEtc| ...)
           28> (|replaceExitEtc| ...)
            29> (|replaceExitEtc,fn| ...)
            <29 (|replaceExitEtc,fn| ...)
           <28 (|replaceExitEtc| ...)
          <27 (|replaceExitEtc,fn| ...)
         <26 (|replaceExitEtc| ...)
        <25 (|replaceExitEtc,fn| ...)
       <24 (|replaceExitEtc| ...)
      <23 (|compSeq1| ...)
     <22 (|compSeq| ...)
    <21 (|compExpression| ...)
   <20 (|comp3| ...)
  <19 (|comp2| ...)
 <18 (|compNoStacking| ...)
<17 (|comp| ...)
17> (|comp| ...)
 18> (|compNoStacking| ...)
  19> (|comp2| ...)
   20> (|comp3| ...)
    21> (|compExpression| ...)
     22> (|comp| ...)
      23> (|compNoStacking| ...)
       24> (|comp2| ...)
        25> (|comp3| ...)
         26> (|compColon| ...)
         <26 (|compColon| ...)
```

```
                        <25 (|comp3| ...)
                       <24 (|comp2| ...)
                      <23 (|compNoStacking| ...)
                     <22 (|comp| ...)
```

In order to explain the compiler we will walk through the compilation of EQ.spad, which handles equations as mathematical objects. We start the system. Most of the structure in Axiom are circular so we have to the `*print-cycle*` to true.

```
root@spiff:/tmp# axiom -nox

(1) -> )lisp (setq *print-circle* t)

Value = T
```

We trace the function we find interesting:

```
(1) -> )lisp (trace |compiler|)

Value = (|compiler|)
```

14.2 The top level compiler command

This is the graph of the functions used for compDefine. The syntax is a graphviz dot file. To generate this graph as a JPEG file, type:

```
tangle v9compDefine.dot bookvol9.pamphlet >v9compdefine.dot
dot -Tjpg v9compiler.dot >v9compiler.jpg
```

— v9compiler.dot —

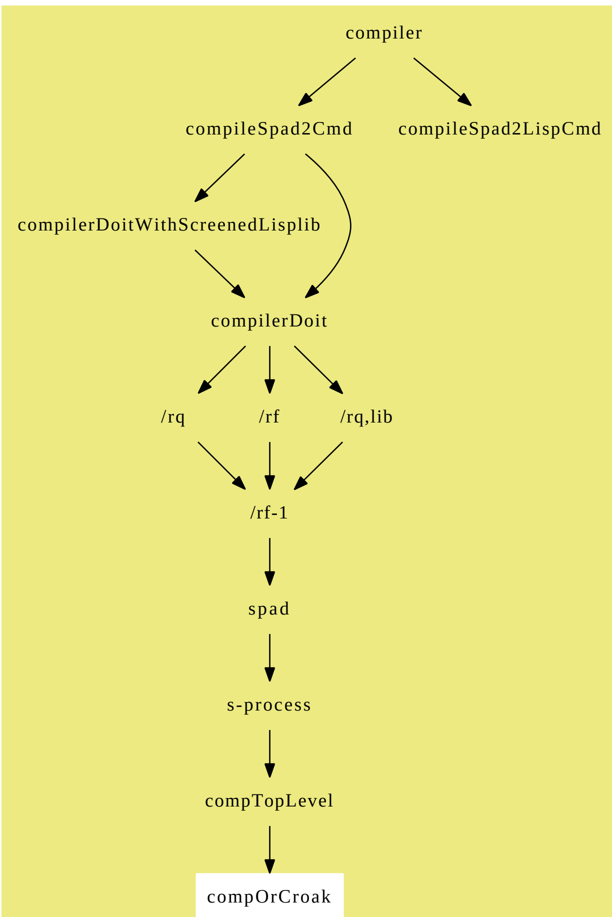
```
digraph pic {
 fontsize=10;
 bgcolor="#ECEA81";
 node [shape=box, color=white, style=filled];

"compiler"                          [color="#ECEA81"]
"compileSpad2Cmd"                   [color="#ECEA81"]
"compileSpad2LispCmd"               [color="#ECEA81"]
"compilerDoitWithScreenedLisplib"   [color="#ECEA81"]
"compilerDoit"                      [color="#ECEA81"]
"/rq"                               [color="#ECEA81"]
"/rf"                               [color="#ECEA81"]
"/rf-1"                             [color="#ECEA81"]
"/rq,lib"                           [color="#ECEA81"]
"spad"                              [color="#ECEA81"]
"s-process"                         [color="#ECEA81"]
"compTopLevel"                      [color="#ECEA81"]
"compOrCroak"                       [color="#FFFFFF"]

"compiler" -> "compileSpad2Cmd"
"compiler" -> "compileSpad2LispCmd"
"compileSpad2Cmd" -> "compilerDoitWithScreenedLisplib"
"compileSpad2Cmd" -> "compilerDoit"
"compilerDoitWithScreenedLisplib" -> "compilerDoit"
```

```
"compilerDoit" -> "/rq"
"compilerDoit" -> "/rf"
"compilerDoit" -> "/rq,lib"
"/rq" -> "/rf-1"
"/rf" -> "/rf-1"
"/rq,lib" -> "/rf-1"
"/rf-1" -> "spad"
"spad" -> "s-process"
"s-process" -> "compTopLevel"
"compTopLevel" -> "compOrCroak"
}
```

————-



14.2.1 defun compiler

We compile the spad file. We can see that the **compiler** function gets a list

```
(1) -> )co EQ

  1> (|compiler| (EQ))
```

In order to find this file, the **pathname** and **pathnameType** functions are used to find the location and pathname to the file. They **pathnameType** function eventually returns the fact that this is a spad source file. Once that is known we call the **compileSpad2Cmd** function with a list containing the full pathname as a string.

```
 1> (|compiler| (EQ))
   2> (|pathname| (EQ))
   <2 (|pathname| #p"EQ")
   2> (|pathnameType| #p"EQ")
     3> (|pathname| #p"EQ")
     <3 (|pathname| #p"EQ")
   <2 (|pathnameType| NIL)
   2> (|pathnameType| "/tmp/EQ.spad")
     3> (|pathname| "/tmp/EQ.spad")
     <3 (|pathname| #p"/tmp/EQ.spad")
   <2 (|pathnameType| "spad")
   2> (|pathnameType| "/tmp/EQ.spad")
     3> (|pathname| "/tmp/EQ.spad")
     <3 (|pathname| #p"/tmp/EQ.spad")
   <2 (|pathnameType| "spad")
   2> (|pathnameType| "/tmp/EQ.spad")
     3> (|pathname| "/tmp/EQ.spad")
     <3 (|pathname| #p"/tmp/EQ.spad")
   <2 (|pathnameType| "spad")
   2> (|compileSpad2Cmd| ("/tmp/EQ.spad"))
```

[compiler helpSpad2Cmd (vol5)]
[compiler selectOptionLC (vol5)]
[compiler pathname (vol5)]
[compiler mergePathnames (vol5)]
[compiler pathnameType (vol5)]
[compiler namestring (vol5)]
[throwKeyedMsg p??]
[findfile p??]
[compileSpad2Cmd p507]
[compileSpadLispCmd p510]
[$newConlist p501]
[$options p??]
[/editfile p??]

compiler : **(CONS SYMBOL NIL)** → **Prompt**

— defun compiler —

```
(defun |compiler| (args)
 "The top level compiler command"
 (let (|$newConlist| ef thefile pathname pathtype)
  (declare (special |$newConlist| |$options| /editfile))
  (setq |$newConlist| nil)
  (if (and (null args) (null |$options|) (null /editfile))
   (|helpSpad2Cmd| '(|compiler|))
   (progn
    (unless args (setq args (cons /editfile nil)))
    (setq pathname (|pathname| args))
    (setq pathtype (|pathnameType| pathname))
```

```
(cond
 ; have we been given a spad file?
 ((string= pathtype "spad")
  (if (null (setq thefile ($findfile pathname '(|spad|))))
     (|throwKeyedMsg| "The spad file %1 is needed but does not exist."
       (cons (namestring pathname) nil))
     (|compileSpad2Cmd| (list thefile))))
 ; have we been given an nrlib?
 ((string= pathtype "nrlib")
  (if (null (setq thefile ($findfile pathname '(|nrlib|))))
    (|throwKeyedMsg| "The nrlib file %1 is needed but does not exist."
     (cons (namestring pathname) nil))
    (|compileSpadLispCmd| (list thefile))))
 (t
  (setq thefile ($findfile pathname '(|spad|)))
  (cond
   ((and thefile (string= (|pathnameType| thefile) "spad"))
    (|compileSpad2Cmd| (list thefile)))
   (t
    (setq ef (|pathname| /editfile))
    (setq ef (|mergePathnames| pathname ef))
    (cond
     ((equal ef pathname)
      (|throwKeyedMsg|
        (format nil "Only AXIOM source files with file extension ~
                     .spad can be compiled.")
         nil))
     (t
      (setq pathname ef)
      (cond
       ((string= (|pathnameType| pathname) "spad")
        (|compileSpad2Cmd| args))
       (t
        (setq thefile ($findfile pathname '(|spad|)))
        (cond
         ((and thefile (string= (|pathnameType| thefile) "spad"))
           (|compileSpad2Cmd| (cons thefile nil)))
         (t (|throwKeyedMsg|
             (format nil "Only AXIOM source files with file extension ~
                          .spad can be compiled.")
              nil)))))))))))))))
```

———

14.2.2 defun compileSpad2Cmd

This function sets up a constructor name (or nil if the whole file is to be compiled) and a set of options for the next layer of compile commands. For example, the call

```
)co dh )constructor DHMATRIX )functions identity
```

will look for the file `dh.spad` and issue a call

```
(|compilerDoitWithScreenedLisplib| (|DenavitHartenbergMatrix|) (|c| |lib|))
```

The argument to this function, as noted above, is a list containing the string pathname to the file.

```
2> (|compileSpad2Cmd| ("/tmp/EQ.spad"))
```

Again we find a lot of redundant work. We finally end up calling **compilerDoit** with a constructed argument list:

```
2> (|compilerDoit| NIL (|rq| |lib|))
```

[compileSpad2Cmd pathname (vol5)]
[compileSpad2Cmd pathnameType (vol5)]
[compileSpad2Cmd namestring (vol5)]
[compileSpad2Cmd updateSourceFiles (vol5)]
[compileSpad2Cmd selectOptionLC (vol5)]
[compileSpad2Cmd terminateSystemCommand (vol5)]
[throwKeyedMsg p??]
[compileSpad2Cmd sayKeyedMsg (vol5)]
[error p??]
[strconc p??]
[object2String p??]
[browserAutoloadOnceTrigger p??]
[spad2AsTranslatorAutoloadOnceTrigger p??]
[compilerDoitWithScreenedLisplib p??]
[compilerDoit p512]
[extendLocalLibdb p429]
[spadPrompt p??]
[$newComp p??]
[$scanIfTrue p??]
[$compileOnlyCertainItems p??]
[$f p??]
[$m p??]
[$QuickLet p??]
[$QuickCode p??]
[$sourceFileTypes p??]
[$InteractiveMode p??]
[$options p??]
[$newConlist p501]
[/editfile p??]

compileSpad2Cmd : **(CONS PathnameString NIL)** → **Prompt**

— defun compileSpad2Cmd —

```
(defun |compileSpad2Cmd| (args)
 (let (|$newComp| |$scanIfTrue|
       |$compileOnlyCertainItems| |$f| |$m| |$QuickLet| |$QuickCode|
       |$sourceFileTypes| |$InteractiveMode| path optlist fun optname
       optargs fullopt constructor)
  (declare (special |$newComp| |$scanIfTrue|
       |$compileOnlyCertainItems| |$f| |$m| |$QuickLet| |$QuickCode|
       |$sourceFileTypes| |$InteractiveMode| /editfile |$options|
       |$newConlist|))
   (setq path (|pathname| args))
   (cond
```

```
((not (string= (|pathnameType| path) "spad"))
  (|throwKeyedMsg|
   (format nil "The old AXIOM system compiler can only compile files ~
               with file extension '.spad'.")
    nil))
((null (probe-file path))
  (|throwKeyedMsg| "The file %1 is needed but does not exist."
   (cons (|namestring| args) nil)))
(t
 (setq /editfile path)
 (|updateSourceFiles| path)
 (|sayKeyedMsg| "Compiling AXIOM source code from %1"
   (list (file-namestring (|namestring| args))))
 (setq optlist '(|break| |constructor| |functions| |library| |lisp|
     |nobreak| |nolibrary| |noquiet| |vartrace| |quiet|))
 (setq |$QuickLet| t)
 (setq |$QuickCode| t)
 (setq fun '(|rq| |lib|))
 (setq |$sourceFileTypes| '("SPAD"))
 (dolist (opt |$options|)
  (setq optname (car opt))
  (setq optargs (cdr opt))
  (setq fullopt (|selectOptionLC| optname optlist nil))
  (case fullopt

   ; library exposes the result
   (|library| (setelt fun 1 '|lib|))

   ; nolibrary compiles but does not expose the reslt
   (|nolibrary| (setelt fun 1 '|nolib|))

   ; quiet surpresses compiler output
   (|quiet| (when (not (eq (elt fun 0) '|c|)) (setelt fun 0 '|rq|)))

   ; noquiet shows compiler output
   (|noquiet| (when (not (eq (elt fun 0) '|c|)) (setelt fun 0 '|rf|)))

   ; compiled code will not cause a break if it fails
   (|nobreak| (setq |$scanIfTrue| t))

   ; compiled code will cause a break if it fails
   (|break| (setq |$scanIfTrue| nil))

   ; allow variable tracing, otherwise lets are inlined
   (|vartrace| (setq |$QuickLet| nil))

   ; compile functions from a domain, e.g.
   ; )co dh )constructor DHMATRIX )functions identity
   (|functions|
    (if (null optargs)
     (|throwKeyedMsg| ")functions requires and argument and you do not give one." nil)
     (setq |$compileOnlyCertainItems| optargs)))

   ; compile a single constructor from a file
```

```
     ; )co dh )constructor DHMATRIX
     (|constructor|
      (if (null optargs)
       (|throwKeyedMsg| ")constructor requires and argument and you do not give one." nil)
       (progn
        (setelt fun 0 '|c|)
        (setq constructor (mapcar #'|unabbrev| optargs)))))
     (t
      (|throwKeyedMsg|
         "%1 is an unknown or unavailable for the )compile command."
       (list (concatenate 'string ")" (string optname)))))))
  (setq |$InteractiveMode| nil)
  (cond
   (|$compileOnlyCertainItems|
    (if (null constructor)
     (|sayKeyedMsg|
      (format nil "The )constructor option to )compile must also be ~
                   specified when the )functions option is used.")
       nil)
     (|compilerDoitWithScreenedLisplib| constructor fun)))
   (t (|compilerDoit| constructor fun)))
  (|extendLocalLibdb| |$newConlist|)
  (|terminateSystemCommand|)
  (|spadPrompt|)))))
```

———-

14.2.3 defun compileSpadLispCmd

[compileSpadLispCmd pathname (vol5)]
[compileSpadLispCmd pathnameType (vol5)]
[compileSpadLispCmd selectOptionLC (vol5)]
[compileSpadLispCmd namestring (vol5)]
[compileSpadLispCmd terminateSystemCommand (vol5)]
[compileSpadLispCmd fnameMake (vol5)]
[compileSpadLispCmd pathnameDirectory (vol5)]
[compileSpadLispCmd pathnameName (vol5)]
[compileSpadLispCmd fnameReadable? (vol5)]
[compileSpadLispCmd localdatabase (vol5)]
[throwKeyedMsg p??]
[object2String p??]
[compileSpadLispCmd sayKeyedMsg (vol5)]
[recompile-lib-file-if-necessary p563]
[spadPrompt p??]
[$options p??]

— defun compileSpadLispCmd —

```
(defun |compileSpadLispCmd| (args)
 (let (path optlist optname optargs beQuiet dolibrary lsp)
  (declare (special |$options|))
  (setq path (|pathname| (|fnameMake| (car args) "code" "lsp")))
```

```
 (cond
  ((null (probe-file path))
    (|throwKeyedMsg| "The file %1 is needed but does not exist."
     (cons (|namestring| args) nil)))
  (t
   (setq optlist '(|quiet| |noquiet| |library| |nolibrary|))
   (setq beQuiet nil)
   (setq dolibrary t)
   (dolist (opt |$options|)
     (setq optname (car opt))
     (setq optargs (cdr opt))
     (case (|selectOptionLC| optname optlist nil)
        (|quiet|     (setq beQuiet t))
        (|noquiet|   (setq beQuiet nil))
        (|library|   (setq dolibrary t))
        (|nolibrary| (setq dolibrary nil))
        (t
         (|throwKeyedMsg|
           "%1 is an unknown or unavailable for the )compile command."
          (list (strconc ")" (|object2String| optname)))))))
   (setq lsp
    (|fnameMake|
     (|pathnameDirectory| path)
     (|pathnameName| path)
     (|pathnameType| path)))
   (cond
    ((|fnameReadable?| lsp)
     (unless beQuiet
      (|sayKeyedMsg| "Compiling Lisp source code from file %1"
       (list (|namestring| lsp))))
      (recompile-lib-file-if-necessary lsp))
    (t
     (|sayKeyedMsg| "The file %1 is needed but does not exist."
      (list (|namestring| lsp)))))
   (cond
    (dolibrary
     (unless beQuiet (|sayKeyedMsg| "Issuing )library command for %1"
                      (list (|pathnameName| path))))
     (localdatabase (list (|pathnameName| (car args))) nil))
    ((null beQuiet)
     (|sayKeyedMsg|
      "The )library system command was not called after compilation." nil))
    (t nil))
   (|terminateSystemCommand|)
   (|spadPrompt|)))))
```

———–

14.2.4 compilerDoitWithScreenedLisplib

compilerDoitWithScreenedLisplib [embed p??]
[rwrite p??]

[compilerDoit p512]
[unembed p??]
[$saveableItems p??]
[$libFile p??]

— defun compilerDoitWithScreenedLisplib —

```
(defun |compilerDoitWithScreenedLisplib| (constructor fun)
 (declare (special |$saveableItems| |$libFile|))
  (embed 'rwrite
          '(lambda (key value stream)
             (cond
               ((and (eq stream |$libFile|)
                     (not (member key |$saveableItems|)))
                value)
               ((not nil) (rwrite key value stream)))))
    (unwind-protect
      (|compilerDoit| constructor fun)
      (unembed 'rwrite)))
```

———–

14.2.5 defun compilerDoit

This trivial function cases on the second argument to decide which combination of operations was requested. For this case we see:

```
(1) -> )co EQ
   Compiling AXIOM source code from file /tmp/EQ.spad using old system
      compiler.
  1> (|compilerDoit| NIL (|rq| |lib|))
    2> (|/RQ,LIB|)

... [snip]...

    <2 (|/RQ,LIB| T)
  <1 (|compilerDoit| T)
(1) ->
```

[compilerDoit /rq (vol5)]
[compilerDoit /rf (vol5)]
[compilerDoit member (vol5)]
[sayBrightly p??]
[opOf p??]
[/RQ,LIB p513]
[$byConstructors p565]
[$constructorsSeen p565]

— defun compilerDoit —

```
(defun |compilerDoit| (constructor fun)
 (let (|$byConstructors| |$constructorsSeen|)
 (declare (special |$byConstructors| |$constructorsSeen|))
```

```
  (cond
   ((equal fun '(|rf| |lib|))   (|/RQ,LIB|))   ; Ignore "noquiet"
   ((equal fun '(|rf| |nolib|)) (/rf))
   ((equal fun '(|rq| |lib|))   (|/RQ,LIB|))
   ((equal fun '(|rq| |nolib|)) (/rq))
   ((equal fun '(|c| |lib|))
    (setq |$byConstructors| (loop for x in constructor collect (|opOf| x)))
    (|/RQ,LIB|)
    (dolist (x |$byConstructors|)
     (unless (|member| x |$constructorsSeen|)
      (|sayBrightly| `(">>> Warning " ,x " was not found")))))))
```

———-

14.2.6 defun /rq

Compile with quiet output [/rf-1 p514]
[echo-meta p??]

— defun /rq —

```
(defun /rq (&rest foo &aux (echo-meta nil))
  (declare (special Echo-Meta) (ignore foo))
  (/rf-1 nil))
```

———-

14.2.7 defun /rf

Compile with noisy output [/rf-1 p514]
[echo-meta p??]

— defun /rf —

```
(defun /rf (&rest foo &aux (echo-meta t))
 (declare (special echo-meta) (ignore foo))
  (/rf-1 nil))
```

———-

14.2.8 defun /RQ,LIB

This function simply calls `/rf-1`.

```
(2) -> )co EQ
   Compiling AXIOM source code from file /tmp/EQ.spad using old system
      compiler.
  1> (|compilerDoit| NIL (|rq| |lib|))
    2> (|/RQ,LIB|)
      3> (/RF-1 NIL)
...[snip]...
```

```
      <3 (/RF-1 T)
    <2 (|/RQ,LIB| T)
  <1 (|compilerDoit| T)
```

[/rf-1 p514]
[/RQ,LIB echo-meta (vol5)]
[$lisplib p??]

— defun /RQ,LIB —

```
(defun |/RQ,LIB| (&rest foo &aux (echo-meta nil) ($lisplib t))
 (declare (special echo-meta $lisplib) (ignore foo))
  (/rf-1 nil))
```

———-

14.2.9 defun /rf-1

Since this function is called with nil we fall directly into the call to the function **spad**:

```
(2) -> )co EQ
   Compiling AXIOM source code from file /tmp/EQ.spad using old system
      compiler.
  1> (|compilerDoit| NIL (|rq| |lib|))
    2> (|/RQ,LIB|)
      3> (/RF-1 NIL)
        4> (SPAD "/tmp/EQ.spad")
...[snip]...
        <4 (SPAD T)
      <3 (/RF-1 T)
    <2 (|/RQ,LIB| T)
  <1 (|compilerDoit| T)
```

[/rf-1 makeInputFilename (vol5)]
[ncINTERPFILE p563]
calls/rf-1spad [/editfile p??]
[echo-meta p??]

— defun /rf-1 —

```
(defun /rf-1 (ignore)
 (declare (ignore ignore))
 (let* ((input-file (makeInputFilename /editfile))
        (type (pathname-type input-file)))
 (declare (special echo-meta /editfile))
 (cond
  ((string= type "lisp") (load input-file))
  ((string= type "input") (|ncINTERPFILE| input-file echo-meta))
  (t (spad input-file)))))
```

———-

14.2.10 defun spad

Here we begin the actual compilation process.

```
  1> (SPAD "/tmp/EQ.spad")
    2> (|makeInitialModemapFrame|)
    <2 (|makeInitialModemapFrame| ((NIL)))
    2> (INIT-BOOT/SPAD-READER)
    <2 (INIT-BOOT/SPAD-READER NIL)
    2> (OPEN "/tmp/EQ.spad" :DIRECTION :INPUT)
    <2 (OPEN #<input stream "/tmp/EQ.spad">)
    2> (INITIALIZE-PREPARSE #<input stream "/tmp/EQ.spad">)
    <2 (INITIALIZE-PREPARSE ")abbrev domain EQ Equation")
    2> (PREPARSE #<input stream "/tmp/EQ.spad">)
   EQ abbreviates domain Equation
    <2 (PREPARSE (# # # # # # # # ...))
    2> (|PARSE-NewExpr|)
    <2 (|PARSE-NewExpr| T)
    2> (S-PROCESS (|where| # #))
...[snip]...
      3> (OPEN "/tmp/EQ.erlib/info" :DIRECTION :OUTPUT)
      <3 (OPEN #<output stream "/tmp/EQ.erlib/info">)
      3> (OPEN #p"/tmp/EQ.nrlib/EQ.lsp")
      <3 (OPEN #<input stream "/tmp/EQ.nrlib/EQ.lsp">)
      3> (OPEN #p"/tmp/EQ.nrlib/EQ.data" :DIRECTION :OUTPUT)
      <3 (OPEN #<output stream "/tmp/EQ.nrlib/EQ.data">)
      3> (OPEN #p"/tmp/EQ.nrlib/EQ.c" :DIRECTION :OUTPUT)
      <3 (OPEN #<output stream "/tmp/EQ.nrlib/EQ.c">)
      3> (OPEN #p"/tmp/EQ.nrlib/EQ.h" :DIRECTION :OUTPUT)
      <3 (OPEN #<output stream "/tmp/EQ.nrlib/EQ.h">)
      3> (OPEN #p"/tmp/EQ.nrlib/EQ.fn" :DIRECTION :OUTPUT)
      <3 (OPEN #<output stream "/tmp/EQ.nrlib/EQ.fn">)
      3> (OPEN #p"/tmp/EQ.nrlib/EQ.o" :DIRECTION :OUTPUT :IF-EXISTS :APPEND)
      <3 (OPEN #<output stream "/tmp/EQ.nrlib/EQ.o">)
      3> (OPEN #p"/tmp/EQ.nrlib/EQ.data")
      <3 (OPEN #<input stream "/tmp/EQ.nrlib/EQ.data">)
      3> (OPEN "/tmp/EQ.nrlib/index.kaf")
      <3 (OPEN #<input stream "/tmp/EQ.nrlib/index.kaf">)
    <2 (S-PROCESS NIL)
  <1 (SPAD T)
  1> (OPEN "temp.text" :DIRECTION :OUTPUT)
  <1 (OPEN #<output stream "temp.text">)
  1> (OPEN "libdb.text")
  <1 (OPEN #<input stream "libdb.text">)
  1> (OPEN "temp.text")
  <1 (OPEN #<input stream "temp.text">)
  1> (OPEN "libdb.text" :DIRECTION :OUTPUT)
  <1 (OPEN #<output stream "libdb.text">)
```

The major steps in this process involve the **preparse** function. (See book volume 5 for more details). The **preparse** function returns a list of pairs of the form: ((linenumber . linestring) .... (linenumber . linestring)) For instance, for the file `EQ.spad`, we get:

```
    <2 (PREPARSE (
 (19 . "Equation(S: Type): public == private where")
```

```
 (20 . " (Ex ==> OutputForm;")
 (21 . "  public ==> Type with")
 (22 . "   (\"=\": (S, S) -> $;")
...[skip]...
 (202 . "        inv eq == [inv lhs eq, inv rhs eq]);")
 (203 . "    if S has ExpressionSpace then")
 (204 . "        subst(eq1,eq2) ==")
 (205 . "           (eq3 := eq2 pretend Equation S;")
 (206 . "            [subst(lhs eq1,eq3),subst(rhs eq1,eq3)])))")))
```

[spad-reader p??]
[spad addBinding (vol5)]
[spad makeInitialModemapFrame (vol5)]
[spad init-boot/spad-reader (vol5)]
[initialize-preparse p81]
[preparse p85]
[PARSE-NewExpr p376]
[pop-stack-1 p495]
[s-process p517]
[ioclear p??]
[spad shut (vol5)]
[$noSubsumption p??]
[$InteractiveFrame p??]
[$InitialDomainsInScope p??]
[$InteractiveMode p??]
[$spad p515]
[$boot p??]
[curoutstream p??]
[*fileactq-apply* p??]
[line p??]
[optionlist p??]
[echo-meta p??]
[/editfile p??]
[*comp370-apply* p??]
[*eof* p??]
[file-closed p??]
[boot-line-stack p??]
[spad-reader p??]

— defun spad —

```
(defun spad (&optional (*spad-input-file* nil) (*spad-output-file* nil)
             &aux (*comp370-apply* #'print-defun)
                  (*fileactq-apply* #'print-defun)
                 ($spad t) ($boot nil) (optionlist nil) (*eof* nil)
                 (file-closed nil) (/editfile *spad-input-file*)
                (|$noSubsumption| |$noSubsumption|) in-stream out-stream)
  (declare (special echo-meta /editfile *comp370-apply* *eof* curoutstream
                    file-closed |$noSubsumption| |$InteractiveFrame|
                    |$InteractiveMode| optionlist
                    boot-line-stack *fileactq-apply* $spad $boot))
  ;; only rebind |$InteractiveFrame| if compiling
```

```
  (progv (if (not |$InteractiveMode|) '(|$InteractiveFrame|))
         (if (not |$InteractiveMode|)
             (list (|addBinding| '|$DomainsInScope|
                     `((fluid . |true|))
                    (|addBinding| '|$Information| nil
                       (|makeInitialModemapFrame|)))))
  (init-boot/spad-reader)
  (unwind-protect
    (progn
      (setq in-stream (if *spad-input-file*
                        (open *spad-input-file* :direction :input)
                         *standard-input*))
      (initialize-preparse in-stream)
      (setq out-stream (if *spad-output-file*
                        (open *spad-output-file* :direction :output)
                         *standard-output*))
      (when *spad-output-file*
         (format out-stream "~&;;; -*- Mode:Lisp; Package:Boot  -*-~%~%")
         (format out-stream "~&~%(IN-PACKAGE \"BOOT\")~%~%"))
      (setq curoutstream out-stream)
      (loop
       (if (or *eof* file-closed) (return nil))
       (catch 'spad_reader
         (if (setq boot-line-stack (preparse in-stream))
             (let ((line (cdar boot-line-stack)))
               (declare (special line))
               (|PARSE-NewExpr|)
               (let ((parseout (pop-stack-1)) )
                 (when parseout
                       (let ((*standard-output* out-stream))
                         (s-process parseout))
                       (format out-stream "~&")))
               )))
      (ioclear in-stream out-stream)))
    (if *spad-input-file* (shut in-stream))
    (if *spad-output-file* (shut out-stream)))
  t))
```

———-

14.2.11 defun Interpreter interface to the compiler

And the **s-process** function which returns a parsed version of the input.

```
    2> (S-PROCESS
(|where|
 (== (|:| (|Equation| (|:| S |Type|)) |public|) |private|)
 (|;|
  (|;|
   (==> |Ex| |OutputForm|)
   (==> |public|
    (|Join| |Type|
     (|with|
```

```
(CATEGORY
 (|Signature| "=" (-> (|,| S S) $))
 (|Signature| |equation| (-> (|,| S S) $))
 (|Signature| |swap| (-> $ $))
 (|Signature| |lhs| (-> $ S))
 (|Signature| |rhs| (-> $ S))
 (|Signature| |map| (-> (|,| (-> S S) $) $))
 (|if| (|has| S (|InnerEvalable| (|,| |Symbol| S)))
  (|Attribute| (|InnerEvalable| (|,| |Symbol| S)))
  NIL)
 (|if| (|has| S |SetCategory|)
  (CATEGORY
   (|Attribute| |SetCategory|)
   (|Attribute| (|CoercibleTo| |Boolean|))
   (|if| (|has| S (|Evalable| S))
    (CATEGORY
     (|Signature| |eval| (-> (|,| $ $) $))
     (|Signature| |eval| (-> (|,| $ (|List| $)) $)))
    NIL))
  NIL)
 (|if| (|has| S |AbelianSemiGroup|)
  (CATEGORY
   (|Attribute| |AbelianSemiGroup|)
   (|Signature| "+" (-> (|,| S $) $))
   (|Signature| "+" (-> (|,| $ S) $)))
   NIL)
 (|if| (|has| S |AbelianGroup|)
  (CATEGORY
   (|Attribute| |AbelianGroup|)
   (|Signature| |leftZero| (-> $ $))
   (|Signature| |rightZero| (-> $ $))
   (|Signature| "-" (-> (|,| S $) $))
   (|Signature| "-" (-> (|,| $ S) $))) NIL)
 (|if| (|has| S |SemiGroup|)
  (CATEGORY
   (|Attribute| |SemiGroup|)
   (|Signature| "*" (-> (|,| S $) $))
   (|Signature| "*" (-> (|,| $ S) $)))
  NIL)
 (|if| (|has| S |Monoid|)
  (CATEGORY
   (|Attribute| |Monoid|)
   (|Signature| |leftOne| (-> $ (|Union| (|,| $ "failed"))))
   (|Signature| |rightOne| (-> $ (|Union| (|,| $ "failed")))))
  NIL)
 (|if| (|has| S |Group|)
  (CATEGORY
   (|Attribute| |Group|)
   (|Signature| |leftOne| (-> $ (|Union| (|,| $ "failed"))))
   (|Signature| |rightOne| (-> $ (|Union| (|,| $ "failed")))))
   NIL)
 (|if| (|has| S |Ring|)
  (CATEGORY
   (|Attribute| |Ring|)
```

```
          (|Attribute| (|BiModule| (|,| S S))))
         NIL)
        (|if| (|has| S |CommutativeRing|)
         (|Attribute| (|Module| S))
         NIL)
        (|if| (|has| S |IntegralDomain|)
         (|Signature| |factorAndSplit| (-> $ (|List| $)))
         NIL)
        (|if| (|has| S (|PartialDifferentialRing| |Symbol|))
         (|Attribute| (|PartialDifferentialRing| |Symbol|))
         NIL)
        (|if| (|has| S |Field|)
         (CATEGORY
          (|Attribute| (|VectorSpace| S))
          (|Signature| "/" (-> (|,| $ $) $))
          (|Signature| |inv| (-> $ $)))
         NIL)
        (|if| (|has| S |ExpressionSpace|)
         (|Signature| |subst| (-> (|,| $ $) $))
         NIL)
    )))))
  (==> |private|
   (|add|
    (|;|
     (|;|
      (|;|
       (|;|
        (|;|
         (|;|
          (|;|
           (|;|
            (|;|
             (|;|
              (|;|
               (|;|
                (|;|
                 (|;|
                  (|;|
                   (|;|
                    (|;|
                     (|;|
                      (|;|
                       (|;|
                        (|;|
                         (|:=| |Rep|
                          (|Record| (|,| (|:| |lhs| S) (|:| |rhs| S))))
                         (|,| |eq1| (|:| |eq2| $)))
                        (|:| |s| S))
                       (|if| (|has| S |IntegralDomain|)
                        (==
                         (|factorAndSplit| |eq|)
                         (|;|
                          (=> (|has| S (|:| |factor| (-> S (|Factored| S))))
```

```
             (|;|
              (|:=| |eq0| (|rightZero| |eq|))
              (COLLECT
               (IN |rcf| (|factors| (|factor| (|lhs| |eq0|))))
               (|construct|
                (|equation| (|,| (|rcf| |factor|) 0))))))
            (|construct| |eq|)))
          NIL))
        (==
         (= (|:| |l| S) (|:| |r| S))
         (|construct| (|,| |l| |r|))))
       (==
        (|equation| (|,| |l| |r|))
        (|construct| (|,| |l| |r|))))
      (== (|lhs| |eqn|) (|eqn| |lhs|)))
     (== (|rhs| |eqn|) (|eqn| |rhs|)))
    (==
     (|swap| |eqn|)
     (|construct| (|,| (|rhs| |eqn|) (|lhs| |eqn|)))))
   (==
    (|map| (|,| |fn| |eqn|))
    (|equation|
     (|,| (|fn| (|eqn| |lhs|)) (|fn| (|eqn| |rhs|))))))
  (|if| (|has| S (|InnerEvalable| (|,| |Symbol| S)))
   (|;|
    (|;|
     (|;|
      (|;|
       (|;| (|:| |s| |Symbol|) (|:| |ls| (|List| |Symbol|)))
       (|:| |x| S))
      (|:| |lx| (|List| S)))
     (==
      (|eval| (|,| (|,| |eqn| |s|) |x|))
      (=
       (|eval| (|,| (|,| (|eqn| |lhs|) |s|) |x|))
       (|eval| (|,| (|,| (|eqn| |rhs|) |s|) |x|)))))
    (==
     (|eval| (|,| (|,| |eqn| |ls|) |lx|))
     (=
      (|eval| (|,| (|,| (|eqn| |lhs|) |ls|) |lx|))
      (|eval| (|,| (|,| (|eqn| |rhs|) |ls|) |lx|)))))
   NIL))
 (|if| (|has| S (|Evalable| S))
  (|;|
   (==
    (|:| (|eval| (|,| (|:| |eqn1| $) (|:| |eqn2| $))) $)
    (=
     (|eval|
      (|,| (|eqn1| |lhs|) (|pretend| |eqn2| (|Equation| S))))
     (|eval|
      (|,| (|eqn1| |rhs|) (|pretend| |eqn2| (|Equation| S))))))
   (==
    (|:|
     (|eval| (|,| (|:| |eqn1| $) (|:| |leqn2| (|List| $)))) $)
```

```
      (=
       (|eval|
        (|,|
         (|eqn1| |lhs|)
         (|pretend| |leqn2| (|List| (|Equation| S)))))
       (|eval|
        (|,|
         (|eqn1| |rhs|)
         (|pretend| |leqn2| (|List| (|Equation| S))))))))
    NIL))
  (|if| (|has| S |SetCategory|)
   (|;|
    (|;|
     (==
      (= |eq1| |eq2|)
      (|and|
       (@ (= (|eq1| |lhs|) (|eq2| |lhs|)) |Boolean|)
       (@ (= (|eq1| |rhs|) (|eq2| |rhs|)) |Boolean|)))
     (==
      (|:| (|coerce| (|:| |eqn| $)) |Ex|)
      (= (|::| (|eqn| |lhs|) |Ex|) (|::| (|eqn| |rhs|) |Ex|))))
    (==
     (|:| (|coerce| (|:| |eqn| $)) |Boolean|)
     (= (|eqn| |lhs|) (|eqn| |rhs|))))
   NIL))
 (|if| (|has| S |AbelianSemiGroup|)
  (|;|
   (|;|
    (==
     (+ |eq1| |eq2|)
     (=
      (+ (|eq1| |lhs|) (|eq2| |lhs|))
      (+ (|eq1| |rhs|) (|eq2| |rhs|))))
    (== (+ |s| |eq2|) (+ (|construct| (|,| |s| |s|)) |eq2|)))
   (== (+ |eq1| |s|) (+ |eq1| (|construct| (|,| |s| |s|)))))
  NIL))
(|if| (|has| S |AbelianGroup|)
 (|;|
  (|;|
   (|;|
    (|;|
     (|;|
      (|;|
       (== (- |eq|) (= (- (|lhs| |eq|)) (- (|rhs| |eq|))))
       (== (- |s| |eq2|) (- (|construct| (|,| |s| |s|)) |eq2|)))
      (== (- |eq1| |s|) (- |eq1| (|construct| (|,| |s| |s|)))))
     (== (|leftZero| |eq|) (= 0 (- (|rhs| |eq|) (|lhs| |eq|)))))
    (== (|rightZero| |eq|) (= (- (|lhs| |eq|) (|rhs| |eq|)) 0)))
   (== 0 (|equation| (|,| (|elt| S 0) (|elt| S 0)))))
  (==
   (- |eq1| |eq2|)
   (=
    (- (|eq1| |lhs|) (|eq2| |lhs|))
    (- (|eq1| |rhs|) (|eq2| |rhs|)))))
```

```
      NIL))
     (|if| (|has| S |SemiGroup|)
      (|;|
       (|;|
        (|;|
         (==
          (* (|:| |eq1| $) (|:| |eq2| $))
          (=
           (* (|eq1| |lhs|) (|eq2| |lhs|))
           (* (|eq1| |rhs|) (|eq2| |rhs|))))
         (==
          (* (|:| |l| S) (|:| |eqn| $))
          (= (* |l| (|eqn| |lhs|)) (* |l| (|eqn| |rhs|)))))
        (==
         (* (|:| |l| S) (|:| |eqn| $))
         (= (* |l| (|eqn| |lhs|)) (* |l| (|eqn| |rhs|)))))
       (==
        (* (|:| |eqn| $) (|:| |l| S))
        (= (* (|eqn| |lhs|) |l|) (* (|eqn| |rhs|) |l|))))
      NIL))
    (|if| (|has| S |Monoid|)
     (|;|
      (|;|
       (|;|
        (== 1 (|equation| (|,| (|elt| S 1) (|elt| S 1))))
        (==
         (|recip| |eq|)
         (|;|
          (|;|
           (=> (|case| (|:=| |lh| (|recip| (|lhs| |eq|))) "failed")
               "failed")
           (=> (|case| (|:=| |rh| (|recip| (|rhs| |eq|))) "failed")
               "failed"))
          (|construct| (|,| (|::| |lh| S) (|::| |rh| S))))))
       (==
        (|leftOne| |eq|)
        (|;|
         (=> (|case| (|:=| |re| (|recip| (|lhs| |eq|))) "failed")
             "failed")
         (= 1 (* (|rhs| |eq|) |re|)))))
      (==
       (|rightOne| |eq|)
       (|;|
        (=> (|case| (|:=| |re| (|recip| (|rhs| |eq|))) "failed")
            "failed")
        (= (* (|lhs| |eq|) |re|) 1))))
     NIL))
   (|if| (|has| S |Group|)
    (|;|
     (|;|
      (==
       (|inv| |eq|)
       (|construct| (|,| (|inv| (|lhs| |eq|)) (|inv| (|rhs| |eq|)))))
      (== (|leftOne| |eq|) (= 1 (* (|rhs| |eq|) (|inv| (|rhs| |eq|))))))
```

```
       (== (|rightOne| |eq|) (= (* (|lhs| |eq|) (|inv| (|rhs| |eq|))) 1)))
      NIL))
    (|if| (|has| S |Ring|)
     (|;|
      (==
       (|characteristic| (|@Tuple|))
       ((|elt| S |characteristic|) (|@Tuple|)))
      (== (* (|:| |i| |Integer|) (|:| |eq| $)) (* (|::| |i| S) |eq|)))
     NIL))
   (|if| (|has| S |IntegralDomain|)
    (==
     (|factorAndSplit| |eq|)
     (|;|
      (|;|
       (=>
        (|has| S (|:| |factor| (-> S (|Factored| S))))
        (|;|
         (|:=| |eq0| (|rightZero| |eq|))
         (COLLECT
          (IN |rcf| (|factors| (|factor| (|lhs| |eq0|))))
          (|construct| (|equation| (|,| (|rcf| |factor|) 0))))))
       (=>
        (|has| S (|Polynomial| |Integer|))
        (|;|
         (|;|
          (|;|
           (|:=| |eq0| (|rightZero| |eq|))
           (==> MF
            (|MultivariateFactorize|
             (|,|
              (|,| (|,| |Symbol| (|IndexedExponents| |Symbol|)) |Integer|)
             (|Polynomial| |Integer|)))))
          (|:=|
           (|:| |p| (|Polynomial| |Integer|))
           (|pretend| (|lhs| |eq0|) (|Polynomial| |Integer|))))
         (COLLECT
          (IN |rcf| (|factors| ((|elt| MF |factor|) |p|)))
          (|construct|
           (|equation| (|,| (|pretend| (|rcf| |factor|) S) 0)))))))
      (|construct| |eq|)))
    NIL))
  (|if| (|has| S (|PartialDifferentialRing| |Symbol|))
   (==
    (|:| (|differentiate| (|,| (|:| |eq| $) (|:| |sym| |Symbol|))) $)
    (|construct|
     (|,|
      (|differentiate| (|,| (|lhs| |eq|) |sym|))
      (|differentiate| (|,| (|rhs| |eq|) |sym|)))))
   NIL))
 (|if| (|has| S |Field|)
  (|;|
   (|;|
    (== (|dimension| (|@Tuple|)) (|::| 2 |CardinalNumber|))
    (==
```

```
        (/ (|:| |eq1| $) (|:| |eq2| $))
        (= (/ (|eq1| |lhs|) (|eq2| |lhs|)) (/ (|eq1| |rhs|) (|eq2| |rhs|)))))
      (==
       (|inv| |eq|)
       (|construct| (|,| (|inv| (|lhs| |eq|)) (|inv| (|rhs| |eq|))))))
     NIL))
   (|if| (|has| S |ExpressionSpace|)
    (==
     (|subst| (|,| |eq1| |eq2|))
     (|;|
      (|:=| |eq3| (|pretend| |eq2| (|Equation| S)))
      (|construct|
       (|,|
        (|subst| (|,| (|lhs| |eq1|) |eq3|))
        (|subst| (|,| (|rhs| |eq1|) |eq3|))))))
    NIL))))))
```

[curstrm p??]
[def-rename p527]
[new2OldLisp p78]
[parseTransform p99]
[postTransform p337]
[displayPreCompilationErrors p490]
[prettyprint p??]
[s-process processInteractive (vol5)]
[compTopLevel p526]
[def-process p??]
[displaySemanticErrors p??]
[terpri p??]
[get-internal-run-time p??]
[$Index p??]
[$macroassoc p??]
[$newspad p??]
[$PolyMode p??]
[$EmptyMode p166]
[$compUniquelyIfTrue p??]
[$currentFunction p??]
[$postStack p??]
[$topOp p??]
[$semanticErrorStack p??]
[$warningStack p??]
[$exitMode p??]
[$exitModeStack p??]
[$returnMode p??]
[$leaveMode p??]
[$leaveLevelStack p??]
[$top-level p??]
[$insideFunctorIfTrue p??]
[$insideExpressionIfTrue p??]
[$insideCoerceInteractiveHardIfTrue p??]

[$insideWhereIfTrue p??]
[$insideCategoryIfTrue p??]
[$insideCapsuleFunctionIfTrue p??]
[$form p??]
[$DomainFrame p??]
[$e p??]
[$EmptyEnvironment p??]
[$genFVar p??]
[$genSDVar p??]
[$VariableCount p??]
[$previousTime p??]
[$LocalFrame p??]
[$Translation p??]
[$TranslateOnly p??]
[$PrintOnly p??]
[$currentLine p??]
[$InteractiveFrame p??]
[curoutstream p??]

— defun s-process —

```
(defun s-process (x)
 (prog ((|$Index| 0)
        ($macroassoc ())
        ($newspad t)
        (|$PolyMode| |$EmptyMode|)
        (|$compUniquelyIfTrue| nil)
        |$currentFunction|
        (|$postStack| nil)
        |$topOp|
        (|$semanticErrorStack| ())
        (|$warningStack| ())
        (|$exitMode| |$EmptyMode|)
        (|$exitModeStack| ())
        (|$returnMode| |$EmptyMode|)
        (|$leaveMode| |$EmptyMode|)
        (|$leaveLevelStack| ())
        $top_level |$insideFunctorIfTrue| |$insideExpressionIfTrue|
        |$insideCoerceInteractiveHardIfTrue| |$insideWhereIfTrue|
        |$insideCategoryIfTrue| |$insideCapsuleFunctionIfTrue| |$form|
        (|$DomainFrame| '((NIL)))
        (|$e| |$EmptyEnvironment|)
        (|$genFVar| 0)
        (|$genSDVar| 0)
        (|$VariableCount| 0)
        (|$previousTime| (get-internal-run-time))
        (|$LocalFrame| '((NIL)))
        (curstrm curoutstream) |$s| |$x| |$m| u)
  (declare (special |$Index| $macroassoc $newspad |$PolyMode| |$EmptyMode|
            |$compUniquelyIfTrue| |$currentFunction| |$postStack| |$topOp|
            |$semanticErrorStack| |$warningStack| |$exitMode| |$exitModeStack|
            |$returnMode| |$leaveMode| |$leaveLevelStack| $top_level
            |$insideFunctorIfTrue| |$insideExpressionIfTrue|
```

```
            |$insideCoerceInteractiveHardIfTrue| |$insideWhereIfTrue|
            |$insideCategoryIfTrue| |$insideCapsuleFunctionIfTrue| |$form|
            |$DomainFrame| |$e| |$EmptyEnvironment| |$genFVar| |$genSDVar|
            |$VariableCount| |$previousTime| |$LocalFrame|
            curstrm |$s| |$x| |$m| curoutstream $traceflag |$Translation|
            |$TranslateOnly| |$PrintOnly| |$currentLine| |$InteractiveFrame|))
   (setq $traceflag t)
   (if (not x) (return nil))
   (if $boot
     (setq x (def-rename (new2OldLisp x)))
     (setq x (|parseTransform| (postTransform x))))
   (when |$TranslateOnly| (return (setq |$Translation| x)))
   (when |$postStack| (|displayPreCompilationErrors|) (return nil))
   (when |$PrintOnly|
        (format t "~S   =====>~%" |$currentLine|)
        (return (prettyprint x)))
   (if (not $boot)
    (if |$InteractiveMode|
      (|processInteractive| x nil)
      (when (setq u (|compTopLevel| x |$EmptyMode| |$InteractiveFrame|))
        (setq |$InteractiveFrame| (third u))))
    (def-process x))
   (when |$semanticErrorStack| (|displaySemanticErrors|))
   (terpri)))
```

———-

14.2.12 defun compTopLevel

[compOrCroak p528]
[$NRTderivedTargetIfTrue p??]
[$killOptimizeIfTrue p??]
[$forceAdd p??]
[$compTimeSum p??]
[$resolveTimeSum p??]
[$packagesUsed p??]
[$envHashTable p??]

— **defun compTopLevel** —

```
(defun |compTopLevel| (form mode env)
 (let (|$NRTderivedTargetIfTrue| |$killOptimizeIfTrue| |$forceAdd|
       |$compTimeSum| |$resolveTimeSum| |$packagesUsed| |$envHashTable|
        t1 t2 t3 val newmode)
 (declare (special |$NRTderivedTargetIfTrue| |$killOptimizeIfTrue|
                   |$forceAdd| |$compTimeSum| |$resolveTimeSum|
                   |$packagesUsed| |$envHashTable| ))
   (setq |$NRTderivedTargetIfTrue| nil)
   (setq |$killOptimizeIfTrue| nil)
   (setq |$forceAdd| nil)
   (setq |$compTimeSum| 0)
   (setq |$resolveTimeSum| 0)
```

```
  (setq |$packagesUsed| NIL)
  (setq |$envHashTable| (make-hashtable 'equal))
  (dolist (u (car (car env)))
   (dolist (v (cdr u))
    (hput |$envHashTable| (cons (car u) (cons (car v) nil)) t)))
  (cond
   ((or (and (consp form) (eq (qfirst form) 'def))
        (and (consp form) (eq (qfirst form) '|where|)
             (progn
               (setq t1 (qrest form))
               (and (consp t1)
               (progn
                (setq t2 (qfirst t1))
                (and (consp t2) (eq (qfirst t2) 'def)))))))
     (setq t3 (|compOrCroak| form mode env))
     (setq val (car t3))
     (setq newmode (second t3))
     (cons val (cons newmode (cons env nil))))
    (t (|compOrCroak| form mode env)))))
```

———-

14.2.13 defun print-defun

[is-console p499]
[print-full p??]
[vmlisp::optionlist p??]
[$PrettyPrint p??]

— defun print-defun —

```
(defun print-defun (name body)
 (let* ((sp (assoc 'vmlisp::compiler-output-stream vmlisp::optionlist))
        (st (if sp (cdr sp) *standard-output*)))
 (declare (special vmlisp::optionlist |$PrettyPrint|))
  (when (and (is-console st) (symbolp name) (fboundp name)
             (not (compiled-function-p (symbol-function name))))
    (compile name))
  (when (or |$PrettyPrint| (not (is-console st)))
        (print-full body st) (force-output st))))
```

———-

14.2.14 defun def-rename

[def-rename p527]

— defun def-rename —

```
(defun def-rename (x)
 (cond
  ((symbolp x)
```

```
   (let ((y (get x 'rename))) (if y (first y) x)))
  ((and (listp x) x)
   (if (eqcar x 'quote)
       x
       (cons (def-rename (first x)) (def-rename (cdr x)))))
  (x)))
```

Given:

```
CohenCategory(): Category == SetCategory with

  kind:(CExpr)->Boolean
  operand:(CExpr,Integer)->CExpr
  numberOfOperand:(CExpr)->Integer
  construct:(CExpr,CExpr)->CExpr
```

the resulting call looks like:

```
 (|compOrCroak|
       (DEF (|CohenCategory|)
        ((|Category|))
        (NIL)
        (|Join|
         (|SetCategory|)
         (CATEGORY |package|
          (SIGNATURE |kind| ((|Boolean|) |CExpr|))
          (SIGNATURE |operand| (|CExpr| |CExpr| (|Integer|)))
          (SIGNATURE |numberOfOperand| ((|Integer|) |CExpr|))
          (SIGNATURE |construct| (|CExpr| |CExpr| |CExpr|)))))
        |$EmptyMode|
        (((
           (|$DomainsInScope|
            (FLUID . |true|)
            (special |$EmptyMode| |$NoValueMode|))))))
```

This compiler call expects the first argument `x` to be a `DEF` form to compile, The second argument, `m`, is the mode. The third argument, `e`, is the environment.

14.2.15 defun compOrCroak

[compOrCroak1 p529]

— defun compOrCroak —

```
(defun |compOrCroak| (form mode env)
  (|compOrCroak1| form mode env nil nil))
```

This results in a call to the inner function with

```
(|compOrCroak1|
```

```
(DEF (|CohenCategory|)
 ((|Category|))
 (NIL)
 (|Join|
  (|SetCategory|)
  (CATEGORY |package|
   (SIGNATURE |kind| ((|Boolean|) |CExpr|))
   (SIGNATURE |operand| (|CExpr| |CExpr| (|Integer|)))
   (SIGNATURE |numberOfOperand| ((|Integer|) |CExpr|))
   (SIGNATURE |construct| (|CExpr| |CExpr| |CExpr|)))))
|$EmptyMode|
((((
    |$DomainsInScope|
    (FLUID . |true|)
    (special |$EmptyMode| |$NoValueMode|)))))
NIL
NIL
|comp|)
```

The inner function augments the environment with information from the compiler stack `$compStack` and `$compErrorMessageStack`. Note that these variables are passed in the argument list so they get preserved on the call stack. The calling function gets called for every inner form so we use this implicit stacking to retain the information.

14.2.16 defun compOrCroak1

[comp p530]
[compOrCroak1,compactify p563]
[stackSemanticError p??]
[mkErrorExpr p??]
[displaySemanticErrors p??]
[say p??]
[displayComp p??]
[userError p??]
[$compStack p??]
[$compErrorMessageStack p??]
[$level p??]
[$s p??]
[$scanIfTrue p??]
[$exitModeStack p??]
[compOrCroak p528]

— defun compOrCroak1 —

```
(defun |compOrCroak1| (form mode env |$compStack| |$compErrorMessageStack|)
 (declare (special |$compStack| |$compErrorMessageStack|))
 (let (td errorMessage)
  (declare (special |$level| |$s| |$scanIfTrue| |$exitModeStack|))
  (cond
   ((setq td (catch '|compOrCroak| (|comp| form mode env))) td)
   (t
     (setq |$compStack|
```

```
    (cons (list form mode env |$exitModeStack|) |$compStack|))
   (setq |$s| (|compOrCroak1,compactify| |$compStack|))
   (setq |$level| (|#| |$s|))
   (setq errorMessage
    (if |$compErrorMessageStack|
     (car |$compErrorMessageStack|)
     '|unspecified error|))
   (cond
    (|$scanIfTrue|
     (|stackSemanticError| errorMessage (|mkErrorExpr| |$level|))
     (list '|failedCompilation| mode env ))
    (t
     (|displaySemanticErrors|)
     (say "****** comp fails at level " |$level| " with expression: ******")
     (|displayComp| |$level|)
     (|userError| errorMessage)))))))
```

———-

14.2.17 defun comp

[compNoStacking p530]
[$compStack p??]
[$exitModeStack p??]

— defun comp —

```
(defun |comp| (form mode env)
 (let (td)
 (declare (special |$compStack| |$exitModeStack|))
  (if (setq td (|compNoStacking| form mode env))
    (setq |$compStack| nil)
    (push (list form mode env |$exitModeStack|) |$compStack|))
  td))
```

———-

14.2.18 defun compNoStacking

`$Representation` is bound in compDefineFunctor, set by doIt. This hack says that when something is undeclared, $ is preferred to the underlying representation – RDJ 9/12/83
[comp2 p531]
[compNoStacking1 p531]
[$compStack p??]
[$Representation p??]
[$EmptyMode p166]

— defun compNoStacking —

```
(defun |compNoStacking| (form mode env)
 (let (td)
```

```
 (declare (special |$compStack| |$Representation| |$EmptyMode|))
  (if (setq td (|comp2| form mode env))
    (if (and (equal mode |$EmptyMode|) (equal (second td) |$Representation|))
      (list (car td) '$ (third td))
      td)
   (|compNoStacking1| form mode env |$compStack|))))
```

———-

14.2.19 defun compNoStacking1

[get p??]
[comp2 p531]
[$compStack p??]

— defun compNoStacking1 —

```
(defun |compNoStacking1| (form mode env |$compStack|)
 (declare (special |$compStack|))
 (let (u td)
  (if (setq u (|get| (if (eq mode '$) '|Rep| mode) '|value| env))
    (if (setq td (|comp2| form (car u) env))
      (list (car td) mode (third td))
      nil)
    nil)))
```

———-

14.2.20 defun comp2

[comp3 p532]
[isDomainForm p319]
[isFunctor p234]
[insert p??]
[opOf p??]
[addDomain p233]
[$bootStrapMode p??]
[$packagesUsed p??]
[$lisplib p??]

— defun comp2 —

```
(defun |comp2| (form mode env)
 (let (tmp1)
  (declare (special |$bootStrapMode| |$packagesUsed| $lisplib))
   (when (setq tmp1 (|comp3| form mode env))
    (destructuring-bind (y mprime env) tmp1
     (when (and $lisplib (|isDomainForm| form env) (|isFunctor| form))
       (setq |$packagesUsed| (|insert| (list (|opOf| form)) |$packagesUsed|)))
     ; isDomainForm test needed to prevent error while compiling Ring
     ; $bootStrapMode-test necessary for compiling Ring in $bootStrapMode
```

```
    (if (and (not (equal mode mprime))
             (or |$bootStrapMode| (|isDomainForm| mprime env)))
      (list y mprime (|addDomain| mprime env))
      (list y mprime env))))))
```

———-

14.2.21 defun comp3

[addDomain p233]
[compWithMappingMode p554]
[compAtom p536]
[getmode p??]
[applyMapping p533]
[compApply p534]
[compColon p271]
[compCoerce p331]
[stringPrefix? p??]
[comp3 pname (vol5)]
[compTypeOf p535]
[compExpression p133]
[comp3 member (vol5)]
[getDomainsInScope p235]
[$e p??]
[$insideCompTypeOf p??]

— defun comp3 —

```
(defun |comp3| (form mode |$e|)
 (declare (special |$e|))
 (let (env op ml u tt tmp1)
 (declare (special |$insideCompTypeOf|))
  (setq |$e| (|addDomain| mode |$e|))
  (setq env |$e|)
  (cond
   ((and (consp mode) (eq (qfirst mode) '|Mapping|))
     (|compWithMappingMode| form mode env))
   ((and (consp mode) (eq (qfirst mode) 'quote)
         (consp (qcdr mode)) (eq (qcddr mode) nil))
    (when (equal form (qcadr mode)) (list form mode |$e|)))
   ((stringp mode)
    (when (and (atom form)
               (or (equal mode form) (equal mode (princ-to-string form))))
     (list mode mode env )))
   ((or (null form) (atom form)) (|compAtom| form mode env))
   (t
    (setq op (car form))
    (cond
     ((and (progn
            (setq tmp1 (|getmode| op env))
            (and (consp tmp1)
                 (eq (qfirst tmp1) '|Mapping|)
```

```
                (progn (setq ml (qrest tmp1)) t)))
           (setq u (|applyMapping| form mode env ml)))
       u)
     ((and (consp op) (eq (qfirst op) 'kappa)
           (consp (qcdr op)) (consp (qcddr op))
           (consp (qcdddr op)) (eq (qcddddr op) nil))
       (|compApply| (qcadr op) (qcaddr op) (qcadddr op) (cdr form) mode env))
     ((eq op '|:|)  (|compColon| form mode env))
     ((eq op '|::|) (|compCoerce| form mode env))
     ((and (null (eq |$insideCompTypeOf| t))
           (|stringPrefix?| "TypeOf" (pname op)))
       (|compTypeOf| form mode env))
      (t
       (setq tt (|compExpression| form mode env))
       (cond
        ((and (consp tt) (consp (qcdr tt)) (consp (qcddr tt))
              (eq (qcdddr tt) nil)
              (null (|member| (qcadr tt) (|getDomainsInScope| (qcaddr tt)))))
         (list (qcar tt) (qcadr tt) (|addDomain| (qcadr tt) (qcaddr tt))))
        (t tt))))))))
```

———-

14.2.22 defun applyMapping

[isCategoryForm p??]
[sublis p??]
[comp p530]
[convert p538]
[member p??]
[get p??]
[getAbbreviation p277]
[encodeItem p173]
[$FormalMapVariableList p249]
[$form p??]
[$op p??]
[$prefix p??]
[$formalArgList p??]

— **defun applyMapping** —

```
(defun |applyMapping| (t0 m e ml)
 (prog (op argl mlp temp1 arglp nprefix opp form pairlis)
 (declare (special |$FormalMapVariableList| |$form| |$op| |$prefix|
                   |$formalArgList|))
  (return
   (progn
    (setq op (car t0))
    (setq argl (cdr t0))
    (cond
     ((not (eql (|#| argl) (1- (|#| ml)))) nil)
     ((|isCategoryForm| (car ml) e)
```

```
(setq pairlis
 (loop for a in argl for v in |$FormalMapVariableList|
  collect (cons v a)))
(setq mlp (sublis pairlis ml))
(setq arglp
 (loop for x in argl for mp in (rest mlp)
  collect (car
           (progn
            (setq temp1 (or (|comp| x mp e) (return '|failed|)))
            (setq e (caddr temp1))
            temp1))))
(when (eq arglp '|failed|) (return nil))
(setq form (cons op arglp))
(|convert| (list form (car mlp) e) m))
(t
(setq arglp
 (loop for x in argl for mp in (rest ml)
  collect (car
           (progn
            (setq temp1 (or (|comp| x mp e) (return '|failed|)))
            (setq e (caddr temp1))
            temp1))))
(when (eq arglp '|failed|) (return nil))
(setq form
 (cond
  ((and (null (|member| op |$formalArgList|))
        (atom op)
        (null (|get| op '|value| e)))
    (setq nprefix
     (or |$prefix| (|getAbbreviation| |$op| (|#| (cdr |$form|)))))
    (setq opp
     (intern (strconc
              (|encodeItem| nprefix) '|;| (|encodeItem| op))))
    (cons opp (append arglp (list '$))))
  (t
   (cons '|call| (cons (list '|applyFun| op) arglp)))))
(setq pairlis
 (loop for a in arglp for v in |$FormalMapVariableList|
  collect (cons v a)))
(|convert| (list form (sublis pairlis (car ml)) e) m))))))
```

———

14.2.23 defun compApply

[comp p530]
[Pair p??]
[removeEnv p??]
[resolve p334]
[AddContour p??]
[$EmptyMode p166]

— defun compApply —

```
(defun |compApply| (sig varl body argl m e)
 (let (temp1 argtl contour code mq bodyq)
 (declare (special |$EmptyMode|))
  (setq argtl
   (loop for x in argl
    collect (progn
             (setq temp1 (|comp| x |$EmptyMode| e))
             (setq e (caddr temp1))
             temp1)))
  (setq contour
   (loop for x in varl
         for mq in (cdr sig)
         for a in argl
    collect
     (|Pair| x
       (list
        (list '|mode| mq)
        (list '|value| (|removeEnv| (|comp| a mq e)))))))
  (setq code
   (cons (list 'lambda varl bodyq)
    (loop for tt in argtl
     collect (car tt))))
  (setq mq (|resolve| m (car sig)))
  (setq bodyq (car (|comp| body mq (|addContour| contour e))))
  (list code mq e)))
```

———-

14.2.24 defun compTypeOf

[eqsubstlist p??]
[get p??]
[put p??]
[comp3 p532]
[$insideCompTypeOf p??]
[$FormalMapVariableList p249]

— defun compTypeOf —

```
(defun |compTypeOf| (form mode env)
 (let (|$insideCompTypeOf| op argl newModemap)
 (declare (special |$insideCompTypeOf| |$FormalMapVariableList|))
  (setq op (car form))
  (setq argl (cdr form))
  (setq |$insideCompTypeOf| t)
  (setq newModemap
    (eqsubstlist argl |$FormalMapVariableList| (|get| op '|modemap| env)))
  (setq env (|put| op '|modemap| newModemap env))
  (|comp3| form mode env)))
```

———-

14.2.25 defun compColonInside

[addDomain p233]
[comp p530]
[coerce p325]
[stackWarning p??]
[opOf p??]
[stackSemanticError p??]
[$newCompilerUnionFlag p??]
[$EmptyMode p166]

— defun compColonInside —

```
(defun |compColonInside| (form mode env mprime)
 (let (mpp warningMessage td tprime)
 (declare (special |$newCompilerUnionFlag| |$EmptyMode|))
    (setq env (|addDomain| mprime env))
    (when (setq td (|comp| form |$EmptyMode| env))
     (cond
      ((equal (setq mpp (second td)) mprime)
       (setq warningMessage
        (list '|:| mprime '| -- should replace by @|))))
     (setq td (list (car td) mprime (third td)))
     (when (setq tprime (|coerce| td mode))
      (cond
       (warningMessage (|stackWarning| warningMessage))
       ((and |$newCompilerUnionFlag| (eq (|opOf| mpp) '|Union|))
        (setq tprime
         (|stackSemanticError|
          (list '|cannot pretend | form '| of mode | mpp '| to mode | mprime )
          nil)))
       (t
        (|stackWarning|
         (list '|:| mprime '| -- should replace by pretend|))))
      tprime))))
```

———-

14.2.26 defun compAtom

[compAtomWithModemap p537]
[get p??]
[modeIsAggregateOf p??]
[compList p540]
[compVector p324]
[convert p538]
[isSymbol p??]
[compSymbol p539]
[primitiveType p539]
[primitiveType p539]
[$Expression p??]

— defun compAtom —

```
(defun |compAtom| (form mode env)
 (prog (tmp1 tmp2 r td tt)
  (declare (special |$Expression|))
   (return
   (cond
    ((setq td
      (|compAtomWithModemap| form mode env (|get| form '|modemap| env))) td)
    ((eq form '|nil|)
     (setq td
      (cond
       ((progn
         (setq tmp1 (|modeIsAggregateOf| '|List| mode env))
         (and (consp tmp1)
              (progn
               (setq tmp2 (qrest tmp1))
               (and (consp tmp2)
                    (eq (qrest tmp2) nil)
                    (progn
                      (setq r (qfirst tmp2)) t)))))
         (|compList| form (list '|List| r) env))
       ((progn
         (setq tmp1 (|modeIsAggregateOf| '|Vector| mode env))
         (and (consp tmp1)
              (progn
               (setq tmp2 (qrest tmp1))
               (and (consp tmp2) (eq (qrest tmp2) nil)
               (progn
                 (setq r (qfirst tmp2)) t)))))
         (|compVector| form (list '|Vector| r) env))))
      (when td (|convert| td mode)))
    (t
     (setq tt
      (cond
       ((|isSymbol| form) (or (|compSymbol| form mode env) (return nil)))
       ((and (equal mode |$Expression|)
             (|primitiveType| form)) (list form mode env ))
       ((stringp form) (list form form env ))
       (t (list form (or (|primitiveType| form) (return nil)) env ))))
     (|convert| tt mode))))))
```

———-

14.2.27 defun compAtomWithModemap

[transImplementation p538]
[modeEqual p335]
[convert p538]
[$NoValueMode p165]

— defun compAtomWithModemap —

```
(defun |compAtomWithModemap| (x m env v)
 (let (tt transimp y)
 (declare (special |$NoValueMode|))
  (cond
   ((setq transimp
    (loop for map in v
     when  ; map is [[.,target],[.,fn]]]
          (and (consp map) (consp (qcar map)) (consp (qcdar map))
               (eq (qcddar map) nil)
               (consp (qcdr map)) (eq (qcddr map) nil)
               (consp (qcadr map)) (consp (qcdadr map))
               (eq (qcddadr map) nil))
     collect
      (list (|transImplementation| x map (qcadadr map)) (qcadar map) env)))
    (cond
     ((setq tt
       (let (result)
        (loop for item in transimp
         when (|modeEqual| m (cadr item))
         do (setq result (or result item)))
        result))
       tt)
     ((eql 1 (|#| (setq transimp
                   (loop for ta in transimp
                    when (setq y (|convert| ta m))
                    collect y))))
       (car transimp))
     ((and (< 0 (|#| transimp)) (equal m |$NoValueMode|))
       (car transimp))
     (t nil))))))
```

———-

14.2.28 defun transImplementation

[genDeltaEntry p??]

— defun transImplementation —

```
(defun |transImplementation| (op map fn)
 (setq fn (|genDeltaEntry| (cons op map)))
 (if (and (consp fn) (eq (qcar fn) 'xlam))
   (cons fn nil)
   (cons '|call| (cons fn nil))))
```

———-

14.2.29 defun convert

[resolve p334]
[coerce p325]

— defun convert —

```
(defun |convert| (td mode)
 (let (res)
  (when (setq res (|resolve| (second td) mode))
   (|coerce| td res))))
```

———-

14.2.30 defun primitiveType

[$DoubleFloat p??]
[$NegativeInteger p??]
[$PositiveInteger p??]
[$NonNegativeInteger p??]
[$String p320]
[$EmptyMode p166]

— defun primitiveType —

```
(defun |primitiveType| (form)
  (declare (special |$DoubleFloat| |$NegativeInteger| |$PositiveInteger|
                    |$NonNegativeInteger| |$String| |$EmptyMode|))
  (cond
    ((null form) |$EmptyMode|)
    ((stringp form) |$String|)
    ((integerp form)
     (cond
       ((eql form 0) |$NonNegativeInteger|)
       ((> form 0) |$PositiveInteger|)
       (t |$NegativeInteger|)))
    ((floatp form) |$DoubleFloat|)
    (t nil)))
```

———-

14.2.31 defun compSymbol

[isFluid p??]
[getmode p??]
[get p??]
[NRTgetLocalIndex p201]
[compSymbol member (vol5)]
[isFunction p??]
[errorRef p??]
[stackMessage p??]
[$Symbol p??]
[$Expression p??]
[$FormalMapVariableList p249]
[$compForModeIfTrue p??]

[$formalArgList p**??**]
[$NoValueMode p165]
[$functorLocalParameters p**??**]
[$Boolean p**??**]
[$NoValue p**??**]

— **defun compSymbol** —

```
(defun |compSymbol| (form mode env)
 (let (v mprime newmode)
  (declare (special |$Symbol| |$Expression| |$FormalMapVariableList|
                    |$compForModeIfTrue| |$formalArgList| |$NoValueMode|
                    |$functorLocalParameters| |$Boolean| |$NoValue|))
   (cond
     ((eq form '|$NoValue|) (list '|$NoValue| |$NoValueMode| env ))
     ((|isFluid| form)
      (setq newmode (|getmode| form env))
      (when newmode (list form (|getmode| form env) env)))
     ((eq form '|true|) (list '(quote t) |$Boolean| env ))
     ((eq form '|false|) (list nil |$Boolean| env ))
     ((or (equal form mode)
          (|get| form '|isLiteral| env)) (list (list 'quote form) form env))
     ((setq v (|get| form '|value| env))
      (cond
        ((member form |$functorLocalParameters|)
         ; s will be replaced by an ELT form in beforeCompile
         (|NRTgetLocalIndex| form)
         (list form (second v) env))
        (t
         ; form has been SETQd
         (list form (second v) env))))
     ((setq mprime (|getmode| form env))
      (cond
        ((and (null (|member| form |$formalArgList|))
              (null (member form |$FormalMapVariableList|))
              (null (|isFunction| form env))
              (null (eq |$compForModeIfTrue| t)))
         (|errorRef| form)))
      (list form mprime env ))
     ((member form |$FormalMapVariableList|)
      (|stackMessage| (list '|no mode found for| form )))
     ((or (equal mode |$Expression|) (equal mode |$Symbol|))
      (list (list 'quote form) mode env ))
     ((null (|isFunction| form env)) (|errorRef| form)))))
```

———-

14.2.32 defun compList

[comp p530]

— **defun compList** —

```
(defun |compList| (form mode env)
 (let (tmp1 tmp2 t0 failed (newmode (second mode)))
  (if (null form)
   (list nil mode env)
   (progn
    (setq t0
     (do ((t3 form (cdr t3)) (x nil))
         ((or (atom t3) failed) (unless failed (nreverse0 tmp2)))
       (setq x (car t3))
       (if (setq tmp1 (|comp| x newmode env))
        (progn
         (setq newmode (second tmp1))
         (setq env (third tmp1))
         (push tmp1 tmp2))
        (setq failed t))))
    (unless failed
     (cons
       (cons 'list (loop for texpr in t0 collect (car texpr)))
       (list (list '|List| newmode) env)))))))
```

———-

14.2.33 defun compForm

[compForm1 p541]
[compArgumentsAndTryAgain p553]
[stackMessageIfNone p??]

— **defun compForm** —

```
(defun |compForm| (form mode env)
 (cond
  ((|compForm1| form mode env))
  ((|compArgumentsAndTryAgain| form mode env))
  (t (|stackMessageIfNone| (list '|cannot compile| form)))))
```

———-

14.2.34 defun compForm1

This function is called if a keyword is found in a compile form but there is no handler listed for the form (See 6). [length p??]
[outputComp p318]
[compOrCroak p528]
[compExpressionList p547]
[coerceable p329]
[comp p530]
[coerce p325]
[compForm2 p548]
[augModemapsFromDomain1 p237]

[getFormModemaps p545]
[nreverse0 p??]
[addDomain p233]
[compToApply p543]
[$NumberOfArgsIfInteger p??]
[$Expression p??]
[$EmptyMode p166]

— defun compForm1 —

```
(defun |compForm1| (form mode env)
 (let (|$NumberOfArgsIfInteger| op argl domain tmp1 opprime ans mmList td
       tmp2 tmp3 tmp4 tmp5 tmp6 tmp7)
 (declare (special |$NumberOfArgsIfInteger| |$Expression| |$EmptyMode|))
    (setq op (car form))
    (setq argl (cdr form))
    (setq |$NumberOfArgsIfInteger| (|#| argl))
    (cond
     ((eq op '|error|)
      (list
       (cons op
        (dolist (x argl (nreverse0 tmp4))
          (setq tmp2 (|outputComp| x env))
          (setq env (third tmp2))
          (push (car tmp2) tmp4)))
       mode env))
     ((and (consp op) (eq (qfirst op) '|elt|)
           (progn
            (setq tmp3 (qrest op))
            (and (consp tmp3)
                 (progn
                  (setq domain (qfirst tmp3))
                  (setq tmp1 (qrest tmp3))
                  (and (consp tmp1)
                       (eq (qrest tmp1) nil)
                       (progn
                        (setq opprime (qfirst tmp1))
                        t))))))
       (cond
        ((eq domain '|Lisp|)
          (list
           (cons opprime
            (dolist (x argl (nreverse tmp7))
             (setq tmp2 (|compOrCroak| x |$EmptyMode| env))
             (setq env (third tmp2))
             (push (car tmp2) tmp7)))
           mode env))
        ((and (equal domain |$Expression|) (eq opprime '|construct|))
          (|compExpressionList| argl mode env))
        ((and (eq opprime 'collect) (|coerceable| domain mode env))
          (when (setq td (|comp| (cons opprime argl) domain env))
           (|coerce| td mode)))
        ((and (consp domain) (eq (qfirst domain) '|Mapping|)
             (setq ans
```

```
              (|compForm2| (cons opprime argl) mode
               (setq env (|augModemapsFromDomain1| domain domain env))
               (dolist (x (|getFormModemaps| (cons opprime argl) env)
                                          (nreverse0 tmp6))
                (when
                 (and (consp x)
                      (and (consp (qfirst x)) (equal (qcaar x) domain)))
                   (push x tmp6))))))
          ans)
        ((setq ans
          (|compForm2| (cons opprime argl) mode
           (setq env (|addDomain| domain env))
           (dolist (x (|getFormModemaps| (cons opprime argl) env)
                   (nreverse0 tmp5))
             (when
               (and (consp x)
                    (and (consp (qfirst x)) (equal (qcaar x) domain)))
               (push x tmp5)))))
          ans)
        ((and (eq opprime '|construct|) (|coerceable| domain mode env))
          (when (setq td (|comp| (cons opprime argl) domain env))
           (|coerce| td mode)))
        (t nil)))
     (t
      (setq env (|addDomain| mode env))
      (cond
       ((and (setq mmList (|getFormModemaps| form env))
             (setq td (|compForm2| form mode env mmList)))
         td)
       (t
         (|compToApply| op argl mode env)))))))
```

———-

14.2.35 defun compToApply

[compNoStacking p530]
[compApplication p544]
[$EmptyMode p166]

— defun compToApply —

```
(defun |compToApply| (op argl m e)
 (let (tt m1)
 (declare (special |$EmptyMode|))
  (setq tt (|compNoStacking| op |$EmptyMode| e))
  (when tt
   (setq m1 (cadr tt))
   (cond
    ((and (consp (car tt)) (eq (qcar (car tt)) 'quote)
          (consp (qcdr (car tt))) (eq (qcddr (car tt)) nil)
          (equal (qcadr (car tt)) m1))
      nil)
```

```
  (t
   (|compApplication| op argl m (caddr tt) tt))))))
```

———-

14.2.36 defun compApplication

[eltForm p??]
[resolve p334]
[coerce p325]
[strconc p??]
[encodeItem p173]
[getAbbreviation p277]
[length p??]
[member p??]
[comp p530]
[isCategoryForm p??]
[$Category p??]
[$formatArgList p??]
[$op p??]
[$form p??]
[$prefix p??]

— defun compApplication —

```
(defun |compApplication| (op argl m env tt)
 (let (argml retm temp1 argTl nprefix opp form eltForm)
  (declare (special |$form| |$op| |$prefix| |$formalArgList| |$Category|))
  (cond
   ((and (consp (cadr tt)) (eq (qcar (cadr tt)) '|Mapping|)
         (consp (qcdr (cadr tt))))
     (setq retm (qcadr (cadr tt)))
     (setq argml (qcddr (cadr tt)))
     (cond
      ((not (eql (|#| argl) (|#| argml))) nil)
      (t
       (setq retm (|resolve| m retm))
       (cond
        ((or (equal retm |$Category|) (|isCategoryForm| retm env))
          nil)
        (t
         (setq argTl
          (loop for x in argl for m in argml
           collect (progn
                    (setq temp1 (or (|comp| x m env) (return '|failed|)))
                    (setq env (caddr temp1))
                    temp1)))
         (cond
          ((eq argTl '|failed|) nil)
          (t
           (setq form
            (cond
```

```
          ((and
            (null
             (or (|member| op |$formalArgList|)
                 (|member| (car tt) |$formalArgList|)))
            (atom (car tt)))
           (setq nprefix
            (or |$prefix| (|getAbbreviation| |$op| (|#| (cdr |$form|)))))
           (setq opp
            (intern
             (strconc (|encodeItem| nprefix) '|;| (|encodeItem| (car tt)))))
           (cons opp
            (append
             (loop for item in argTl collect (car item))
             (list '$))))
          (t
           (cons '|call|
            (cons  (list '|applyFun| (car tt))
             (loop for item in argTl collect (car item)))))))
           (|coerce| (list form retm env) (|resolve| retm m))))))))
((eq op '|elt|) nil)
(t
 (setq eltForm (cons '|elt| (cons op argl)))
 (|comp| eltForm m env)))))
```

———-

14.2.37 defun getFormModemaps

[getFormModemaps p545]
[nreverse0 p??]
[get p??]
[eltModemapFilter p546]
[last p??]
[length p??]
[stackMessage p??]
[$insideCategoryPackageIfTrue p??]

— defun getFormModemaps —

```
(defun |getFormModemaps| (form env)
 (let (op argl domain op1 modemapList nargs finalModemapList)
 (declare (special |$insideCategoryPackageIfTrue|))
  (setq op (car form))
  (setq argl (cdr form))
  (cond
   ((and (consp op) (eq (qfirst op) '|elt|) (CONSP (qrest op))
         (consp (qcddr op)) (eq (qcdddr op) nil))
     (setq op1 (third op))
     (setq domain (second op))
     (loop for x in (|getFormModemaps| (cons op1 argl) env)
      when (and (consp x) (consp (qfirst x)) (equal (qcaar x) domain))
      collect x))
```

```
   ((null (atom op)) nil)
   (t
    (setq modemapList (|get| op '|modemap| env))
    (when |$insideCategoryPackageIfTrue|
     (setq modemapList
      (loop for x in modemapList
       when (and (consp x) (consp (qfirst x)) (not (eq (qcaar x) '$)))
       collect x)))))
  (cond
   ((eq op '|elt|)
    (setq modemapList (|eltModemapFilter| (|last| argl) modemapList env)))
   ((eq op '|setelt|)
    (setq modemapList (|seteltModemapFilter| (CADR argl) modemapList env))))
   (setq nargs (|#| argl))
   (setq finalModemapList
    (loop for mm in modemapList
     when (equal (|#| (cddar mm)) nargs)
     collect mm))
 (when (and modemapList (null finalModemapList))
  (|stackMessage|
   (list '|no modemap for| op '|with | nargs '| arguments|)))
  finalModemapList))
```

———-

14.2.38 defun eltModemapFilter

[isConstantId p??]
[stackMessage p??]

— defun eltModemapFilter —

```
(defun |eltModemapFilter| (name mmList env)
 (let (z)
  (if (|isConstantId| name env)
   (cond
    ((setq z
      (loop for mm in mmList
       when (and (consp mm) (consp (qfirst mm)) (consp (qcdar mm))
                 (consp (qcddar mm))
                 (consp (qcdddar mm))
                 (equal (fourth (first mm)) name))
       collect mm))
      z)
    (t
     (|stackMessage|
      (list '|selector variable: | name '| is undeclared and unbound|))
     nil))
    mmList)))
```

———-

14.2.39 defun seteltModemapFilter

[isConstantId p??]
[stackMessage p??]

— defun seteltModemapFilter —

```
(defun |seteltModemapFilter| (name mmList env)
 (let (z)
  (if (|isConstantId| name env)
   (cond
    ((setq z
      (loop for mm in mmList
       when (equal (car (cdddar mm)) name)
       collect mm))
      z)
     (t
      (|stackMessage|
       (list '|selector variable: | name '| is undeclared and unbound|))
      nil))
   mmList)))
```

———-

14.2.40 defun compExpressionList

[nreverse0 p??]
[comp p530]
[convert p538]
[$Expression p??]

— defun compExpressionList —

```
(defun |compExpressionList| (argl m env)
 (let (tmp1 tlst)
 (declare (special |$Expression|))
  (setq tlst
   (prog (result)
    (return
     (do ((tmp2 argl (cdr tmp2)) (x nil))
         ((or (atom tmp2)) (nreverse0 result))
      (setq x (car tmp2))
      (setq result
       (cons
        (progn
         (setq tmp1 (or (|comp| x |$Expression| env) (return '|failed|)))
         (setq env (third tmp1))
         tmp1)
        result))))))
  (unless (eq tlst '|failed|)
   (|convert|
    (list (cons 'list
     (prog (result)
```

```
     (return
      (do ((tmp3 tlst (cdr tmp3)) (y nil))
          ((or (atom tmp3)) (nreverse0 result))
       (setq y (car tmp3))
       (setq result (cons (car y) result))))))
     |$Expression| env)
   m))))
```

———-

14.2.41 defun compForm2

[take p??]
[length p??]
[nreverse0 p??]
[sublis p??]
[assoc p??]
[PredImplies p??]
[isSimple p??]
[compUniquely p553]
[compFormPartiallyBottomUp p552]
[compForm3 p550]
[$EmptyMode p166]
[$TriangleVariableList p??]

— defun compForm2 —

```
(defun |compForm2| (form mode env modemapList)
 (let (op argl sargl aList dc cond nsig v ncond deleteList newList td tl
      partialModeList tmp1 tmp2 tmp3 tmp4 tmp5 tmp6 tmp7)
 (declare (special |$EmptyMode| |$TriangleVariableList|))
  (setq op (car form))
  (setq argl (cdr form))
  (setq sargl (take (|#| argl) |$TriangleVariableList|))
  (setq aList (mapcar #'(lambda (x y) (cons x y)) sargl argl))
  (setq modemaplist (sublis aList modemapList))
  ; now delete any modemaps that are subsumed by something else, provided
  ; the conditions are right (i.e. subsumer true whenever subsumee true)
  (dolist (u modemapList)
   (cond
    ((and (consp u)
          (progn
           (setq tmp6 (qfirst u))
           (and (consp tmp6) (progn (setq dc (qfirst tmp6)) t)))
          (progn
           (setq tmp7 (qrest u))
           (and (consp tmp7) (eq (qrest tmp7) nil)
                (progn
                 (setq tmp1 (qfirst tmp7))
                 (and (consp tmp1)
                      (progn
                       (setq cond (qfirst tmp1))
```

```
                     (setq tmp2 (qrest tmp1))
                     (and (consp tmp2) (eq (qrest tmp2) nil)
                          (progn
                           (setq tmp3 (qfirst tmp2))
                           (and (consp tmp3) (eq (qfirst tmp3) '|Subsumed|)
                                (progn
                                 (setq tmp4 (qrest tmp3))
                                 (and (consp tmp4)
                                      (progn
                                       (setq tmp5 (qrest tmp4))
                                       (and (consp tmp5)
                                            (eq (qrest tmp5) nil)
                                            (progn
                                             (setq nsig (qfirst tmp5))
                                             t)))))))))))))
        (setq v (|assoc| (cons dc nsig) modemapList))
        (consp v)
        (progn
         (setq tmp6 (qrest v))
         (and (consp tmp6) (eq (qrest tmp6) nil)
              (progn
               (setq tmp7 (qfirst tmp6))
               (and (consp tmp7)
                    (progn
                     (setq ncond (qfirst tmp7))
                     t))))))
    (setq deleteList (cons u deleteList))
    (unless  (|PredImplies| ncond cond)
       (setq newList (push `(,(car u) (,cond (elt ,dc nil))) newList))))))
(when deleteList
 (setq modemapList
  (remove-if #'(lambda (x) (member x deletelist)) modemapList)))
; it is important that subsumed ops (newList) be considered last
(when newList (setq modemapList (append modemapList newList)))
(setq tl
 (loop for x in argl
       while (and (|isSimple| x)
                  (setq td (|compUniquely| x |$EmptyMode| env)))
       collect td
       do (setq env (third td))))
(cond
 ((some #'identity tl)
   (setq partialModeList (loop for x in tl collect (when x (second x))))
   (or
     (|compFormPartiallyBottomUp| form mode env modemapList partialModeList)
     (|compForm3| form mode env modemapList)))
 (t (|compForm3| form mode env modemapList)))))
```

———

14.2.42 defun compForm3

[compFormWithModemap p??]
[compUniquely p553]
[$compUniquelyIfTrue p??]

— defun compForm3 —

```
(defun |compForm3| (form mode env modemapList)
 (let (op argl mml tt)
 (declare (special |$compUniquelyIfTrue|))
  (setq op (car form))
  (setq argl (cdr form))
  (setq tt
   (let (result)
    (maplist #'(lambda (mlst)
     (setq result (or result
       (|compFormWithModemap| form mode env (car (setq mml mlst))))))
     modemapList)
    result))
   (when |$compUniquelyIfTrue|
    (if (let (result)
         (mapcar #'(lambda (mm)
            (setq result (or result (|compFormWithModemap| form mode env mm))))
           (rest mml))
          result)
     (throw '|compUniquely| nil)
     tt))
  tt))
```

———

14.2.43 defun compFocompFormWithModemap

[isCategoryForm p??]
[isFunctor p234]
[substituteIntoFunctorModemap p552]
[listOfSharpVars p??]
[coerceable p329]
[compApplyModemap p239]
[isCategoryForm p??]
[identp p??]
[get p??]
[last p??]
[convert p538]
[$Category p??]
[$FormalMapVariableList p249]

— defun compFormWithModemap —

```
(defun |compFormWithModemap| (form m env modemap)
 (prog (op argl sv target cexpr targetp map temp1 f transimp sl mp formp z c
```

```
      xp ep tt)
 (declare (special |$Category| |$FormalMapVariableList|))
 (return
  (progn
   (setq op (car form))
   (setq argl (cdr form))
   (setq map (car modemap))
   (setq target (cadar modemap))
   (when (and (|isCategoryForm| target env) (|isFunctor| op))
     (setq temp1 (or (|substituteIntoFunctorModemap| argl modemap env)
                     (return nil)))
     (setq modemap (car temp1))
     (setq env (cadr temp1))
     (setq map (car modemap))
     (setq target (cadar modemap))
     (setq cexpr (cdr modemap))
    modemap)
   (setq sv (|listOfSharpVars| map))
   (when sv
     (loop for x in argl for ss in |$FormalMapVariableList|
      do (when (|member| ss sv)
           (setq modemap (subst x ss modemap :test #'equal))
           (setq map (car modemap))
           (setq target (cadar modemap))
           (setq cexpr (cdr modemap))
          modemap)))
   (cond
    ((null (setq targetp (|coerceable| target m env))) nil)
    (t
     (setq map (cons targetp (cdr map)))
     (setq temp1 (or (|compApplyModemap| form modemap env nil)
                     (return nil)))
     (setq f (car temp1))
     (setq transimp (cadr temp1))
     (setq sl (caddr temp1))
     (setq mp (sublis sl (elt map 1)))
     (setq xp
      (progn
       (setq formp (cons f (loop for tt in transimp collect (car tt))))
       (cond
        ((or (equal mp |$Category|) (|isCategoryForm| mp env)) formp)
        ((and (eq op '|elt|) (consp f) (eq (qcar f) 'xlam)
              (identp (car argl))
              (setq c (|get| (car argl) '|condition| env))
              (consp c) (eq (qcdr c) nil)
              (consp (qcar c)) (eq (qcaar c) '|case|)
              (consp (qcdar c)) (equal (qcadar c) z)
              (consp (qcddar c)) (eq (qcdr (qcddar c)) nil)
              (or (and (consp (qcaddar c))
                       (eq (qcar (qcaddar c)) '|:|)
                       (consp (qcdr (qcaddar c)))
                       (equal (qcadr (qcaddar c)) (cadr argl))
                       (consp (qcddr (qcaddar c)))
                       (eq (qcdddr (qcaddar c)) nil)
```

```
                      (equal (qcaddr (qcaddar c)) m))
                 (eq (qcaddar c) (cadr argl))))
         (list 'cdr (car argl)))
       (t (cons '|call| formp)))))
    (setq ep
     (if transimp
      (caddr (|last| transimp))
      env))
    (setq tt (list xp mp ep))
    (|convert| tt m)))))))
```

———-

14.2.44 defun substituteIntoFunctorModemap

[keyedSystemError p??]
[eqsubstlist p??]
[compOrCroak p528]
[sublis p??]

— defun substituteIntoFunctorModemap —

```
(defun |substituteIntoFunctorModemap| (argl modemap env)
 (let (dc sig tmp1 tl substitutionList)
  (setq dc (caar modemap))
  (setq sig (cdar modemap))
  (cond
   ((not (eql (|#| dc) (|#| sig)))
     (|keyedSystemError|
          "Unexpected error or improper call to system function %1: %2"
      (list "substituteIntoFunctorModemap" "Incompatible maps")))
   ((equal (|#| argl) (|#| (cdr sig)))
    (setq sig (eqsubstlist argl (cdr dc) sig))
    (setq tl
     (loop for a in argl for m in (rest sig)
      collect (progn
               (setq tmp1 (|compOrCroak| a m env))
               (setq env (caddr tmp1))
               tmp1)))
    (setq substitutionList
     (loop for x in (rest dc) for tt in tl
      collect (cons x (car tt))))
     (list (sublis substitutionList modemap) env))
   (t nil))))
```

———-

14.2.45 defun compFormPartiallyBottomUp

[compForm3 p550]
[compFormMatch p553]

— defun compFormPartiallyBottomUp —

```
(defun |compFormPartiallyBottomUp| (form mode env modemapList partialModeList)
 (let (mmList)
  (when (setq mmList (loop for mm in modemapList
                      when (|compFormMatch| mm partialModeList)
                      collect mm))
   (|compForm3| form mode env mmList))))
```

———-

14.2.46 defun compFormMatch

— defun compFormMatch —

```
(defun |compFormMatch| (mm partialModeList)
 (labels (
  (ismatch (a b)
   (cond
    ((null b) t)
    ((null (car b)) (|compFormMatch,match| (cdr a) (cdr b)))
    ((and (equal (car a) (car b)) (ismatch (cdr a) (cdr b)))))))
  (and (consp mm) (consp (qfirst mm)) (consp (qcdar mm))
       (ismatch (qcddar mm) partialModeList))))
```

———-

14.2.47 defun compUniquely

[compUniquely p553]
[comp p530]
[$compUniquelyIfTrue p??]

— defun compUniquely —

```
(defun |compUniquely| (x m env)
 (let (|$compUniquelyIfTrue|)
 (declare (special |$compUniquelyIfTrue|))
  (setq |$compUniquelyIfTrue| t)
  (catch '|compUniquely| (|comp| x m env))))
```

———-

14.2.48 defun compArgumentsAndTryAgain

[comp p530]
[compForm1 p541]
[$EmptyMode p166]

— defun compArgumentsAndTryAgain —

```
(defun |compArgumentsAndTryAgain| (form mode env)
 (let (argl tmp1 a tmp2 tmp3 u)
 (declare (special |$EmptyMode|))
  (setq argl (cdr form))
  (cond
   ((and (consp form) (eq (qfirst form) '|elt|)
         (progn
           (setq tmp1 (qrest form))
           (and (consp tmp1)
                (progn
                 (setq a (qfirst tmp1))
                 (setq tmp2 (qrest tmp1))
                 (and (consp tmp2) (eq (qrest tmp2) nil))))))
    (when (setq tmp3 (|comp| a |$EmptyMode| env))
      (setq env (third tmp3))
      (|compForm1| form mode env)))
   (t
     (setq u
      (dolist (x argl)
       (setq tmp3 (or (|comp| x |$EmptyMode| env) (return '|failed|)))
       (setq env (third tmp3))
       tmp3))
     (unless (eq u '|failed|)
       (|compForm1| form mode env))))))
```

———-

14.2.49 defun compWithMappingMode

[compWithMappingMode1 p554]
[$formalArgList p??]

— defun compWithMappingMode —

```
(defun |compWithMappingMode| (form mode oldE)
  (declare (special |$formalArgList|))
  (|compWithMappingMode1| form mode oldE |$formalArgList|))
```

———-

14.2.50 defun compWithMappingMode1

[isFunctor p234]
[get p??]
[extendsCategoryForm p??]
[compLambda p299]
[stackAndThrow p??]
[take p??]

[compMakeDeclaration p561]
[hasFormalMapVariable p560]
[comp p530]
[extractCodeAndConstructTriple p559]
[optimizeFunctionDef p212]
[comp-tran p??]
[freelist p562]
[$formalArgList p??]
[$killOptimizeIfTrue p??]
[$funname p??]
[$funnameTail p??]
[$QuickCode p??]
[$EmptyMode p166]
[$FormalMapVariableList p249]
[$CategoryFrame p??]
[$formatArgList p??]

— defun compWithMappingMode1 —

```
(defun |compWithMappingMode1| (form mode oldE |$formalArgList|)
 (declare (special |$formalArgList|))
  (prog (|$killOptimizeIfTrue| $funname $funnameTail mprime sl tmp1 tmp2
         tmp3 tmp4 tmp5 tmp6 target argModeList nx oldstyle ress vl1 vl e tt
             u frees i scode locals body vec expandedFunction fname uu)
  (declare (special |$killOptimizeIfTrue| $funname $funnameTail
                    |$QuickCode| |$EmptyMode| |$FormalMapVariableList|
                    |$CategoryFrame| |$formatArgList|))
   (return
    (seq
     (progn
      (setq mprime (second mode))
      (setq sl (cddr mode))
      (setq |$killOptimizeIfTrue| t)
      (setq e oldE)
      (cond
       ((|isFunctor| form)
        (cond
         ((and (progn
                (setq tmp1 (|get| form '|modemap| |$CategoryFrame|))
                (and (consp tmp1)
                     (progn
                      (setq tmp2 (qfirst tmp1))
                      (and (consp tmp2)
                           (progn
                            (setq tmp3 (qfirst tmp2))
                            (and (consp tmp3)
                                 (progn
                                  (setq tmp4 (qrest tmp3))
                                  (and (consp tmp4)
                                       (progn
                                        (setq target (qfirst tmp4))
                                        (setq argModeList (qrest tmp4))
                                        t)))))
```

```
                    (progn
                     (setq tmp5 (qrest tmp2))
                     (and (consp tmp5) (eq (qrest tmp5) nil)))))))
         (prog (t1)
          (setq t1 t)
          (return
           (do ((t2 nil (null t1))
                (t3 argModeList (cdr t3))
                (newmode nil)
                (t4 sl (cdr t4))
                (s nil))
              ((or t2 (atom t3)
                   (progn (setq newmode (car t3)) nil)
                   (atom t4)
                   (progn (setq s (car t4)) nil))
                  t1)
             (seq (exit
               (setq t1
                (and t1 (|extendsCategoryForm| '$ s newmode))))))))
                  (|extendsCategoryForm| '$ target mprime))
         (return (list form mode e )))
   (t nil)))
  (t
   (when (stringp form) (setq form (intern form)))
   (setq ress nil)
   (setq oldstyle t)
   (cond
    ((and (consp form)
          (eq (qfirst form) '+->)
          (progn
           (setq tmp1 (qrest form))
           (and (consp tmp1)
                (progn
                 (setq vl (qfirst tmp1))
                 (setq tmp2 (qrest tmp1))
                 (and (consp tmp2)
                      (eq (qrest tmp2) nil)
                      (progn (setq nx (qfirst tmp2)) t))))))
      (setq oldstyle nil)
      (cond
       ((and (consp vl) (eq (qfirst vl) '|:|))
         (setq ress (|compLambda| form mode oldE))
         ress)
       (t
        (setq vl
         (cond
          ((and (consp vl)
                (eq (qfirst vl) '|@Tuple|)
                (progn (setq vl1 (qrest vl)) t))
            vl1)
          (t vl)))
        (setq vl
         (cond
          ((symbolp vl) (cons vl nil))
```

```
                ((and
                  (listp vl)
                  (prog (t5)
                   (setq t5 t)
                   (return
                    (do ((t7 nil (null t5))
                         (t6 vl (cdr t6))
                         (v nil))
                       ((or t7 (atom t6) (progn (setq v (car t6)) nil)) t5)
                      (seq
                       (exit
                        (setq t5 (and t5 (symbolp v)))))))))
                  vl)
                (t
                 (|stackAndThrow| (cons '|bad +-> arguments:| (list vl ))))))
              (setq |$formatArgList| (append vl |$formalArgList|))
              (setq form nx))))
         (t
          (setq vl (take (|#| sl) |$FormalMapVariableList|))))
        (cond
         (ress ress)
         (t
          (do ((t8 sl (cdr t8)) (m nil) (t9 vl (cdr t9)) (v nil))
              ((or (atom t8)
                   (progn (setq m (car t8)) nil)
                   (atom t9)
                   (progn (setq v (car t9)) nil))
                nil)
              (seq (exit (progn
               (setq tmp6
                (|compMakeDeclaration| (list '|:| v m ) |$EmptyMode| e))
               (setq e (third tmp6))
               tmp6))))
          (cond
           ((and oldstyle
                 (null (null vl))
                 (null (|hasFormalMapVariable| form vl)))
            (return
             (progn
              (setq tmp6 (or (|comp| (cons form vl) mprime e) (return nil)))
              (setq u (car tmp6))
              (|extractCodeAndConstructTriple| u mode oldE))))
           ((and (null vl) (setq tt (|comp| (cons form nil) mprime e)))
            (return
             (progn
               (setq u (car tt))
               (|extractCodeAndConstructTriple| u mode oldE))))
           (t
            (setq tmp6 (or (|comp| form mprime e) (return nil)))
            (setq u (car tmp6))
            (setq uu (|optimizeFunctionDef| `(nil (lambda ,vl ,u))))
;  --  At this point, we have a function that we would like to pass.
;  --  Unfortunately, it makes various free variable references outside
;  --  itself.  So we build a mini-vector that contains them all, and
```

```
;  --  pass this as the environment to our inner function.
            (setq $funname nil)
            (setq $funnameTail (list nil))
            (setq expandedFunction (comp-tran (second uu)))
            (setq frees (freelist expandedFunction vl nil e))
            (setq expandedFunction
             (cond
              ((eql (|#| frees) 0)
               (cons 'lambda (cons (append vl (list '$$))
                                   (cddr expandedFunction))))
              ((eql (|#| frees) 1)
               (setq vec (caar frees))
                (cons 'lambda (cons (append vl (list vec))
                                   (cddr expandedFunction))))
              (t
               (setq scode nil)
               (setq vec nil)
               (setq locals nil)
               (setq i -1)
               (do ((t0 frees (cdr t0)) (v nil))
                   ((or (atom t0) (progn (setq v (car t0)) nil)) nil)
                 (seq
                  (exit
                   (progn
                    (setq i (plus i 1))
                    (setq vec (cons (car v) vec))
                    (setq scode
                     (cons
                      (cons 'setq
                       (cons (car v)
                        (cons
                         (cons
                          (cond
                           (|$QuickCode| 'qrefelt)
                           (t 'elt))
                          (cons '$$ (cons i nil)))
                         nil)))
                       scode))
                    (setq locals (cons (car v) locals))))))
               (setq body (cddr expandedFunction))
               (cond
                (locals
                 (cond
                  ((and (consp body)
                        (progn
                         (setq tmp1 (qfirst body))
                         (and (consp tmp1)
                              (eq (qfirst tmp1) 'declare))))
                    (setq body
                     (cons (car body)
                      (cons
                       (cons 'prog
                        (cons locals
                         (append scode
```

```
                    (cons
                     (cons 'return
                      (cons
                       (cons 'progn
                        (cdr body))
                       nil))
                     nil))))
                  nil))))
            (t
             (setq body
              (cons
               (cons 'prog
                (cons locals
                 (append scode
                  (cons
                   (cons 'return
                    (cons
                     (cons 'progn body)
                     nil))
                   nil))))
               nil))))))
         (setq vec (cons 'vector (nreverse vec)))
         (cons 'lambda (cons (append vl (list '$$)) body)))))
      (setq fname (list 'closedfn expandedFunction))
      (setq uu
       (cond
        (frees (list 'cons fname vec))
        (t (list 'list fname))))
      (list uu mode oldE)))))))))))
```

———-

14.2.51 defun extractCodeAndConstructTriple

— defun extractCodeAndConstructTriple —

```
(defun |extractCodeAndConstructTriple| (form mode oldE)
 (let (tmp1 a fn op env)
  (cond
   ((and (consp form) (eq (qfirst form) '|call|)
         (progn
          (setq tmp1 (qrest form))
          (and (consp tmp1)
               (progn (setq fn (qfirst tmp1)) t))))
    (cond
     ((and (consp fn) (eq (qfirst fn) '|applyFun|)
           (progn
            (setq tmp1 (qrest fn))
            (and (consp tmp1) (eq (qrest tmp1) nil)
                 (progn (setq a (qfirst tmp1)) t))))
            (setq fn a)))
    (list fn mode oldE))
```

```
  (t
   (setq op (car form))
   (setq env (car (reverse (cdr form))))
   (list (list 'cons  (list 'function op) env) mode oldE)))))
```

———-

14.2.52 defun hasFormalMapVariable

[hasFormalMapVariable ScanOrPairVec (vol5)]
[$formalMapVariables p??]

— defun hasFormalMapVariable —

```
(defun |hasFormalMapVariable| (x vl)
 (let (|$formalMapVariables|)
 (declare (special |$formalMapVariables|))
 (when (setq |$formalMapVariables| vl)
  (|ScanOrPairVec| #'(lambda (y) (member y |$formalMapVariables|)) x))))
```

———-

14.2.53 defun argsToSig

— defun argsToSig —

```
(defun |argsToSig| (args)
 (let (tmp1 v tmp2 tt sig1 arg1 bad)
  (cond
   ((and (consp args) (eq (qfirst args) '|:|)
         (progn
          (setq tmp1 (qrest args))
          (and (consp tmp1)
               (progn
                (setq v (qfirst tmp1))
                (setq tmp2 (qrest tmp1))
                (and (consp tmp2)
                     (eq (qrest tmp2) nil)
                     (progn
                      (setq tt (qfirst tmp2))
                      t))))))
    (list (list v) (list tt)))
   (t
    (setq sig1 nil)
    (setq arg1 nil)
    (setq bad nil)
    (dolist (arg args)
      (cond
       ((and (consp arg) (eq (qfirst arg) '|:|)
             (progn
              (setq tmp1 (qrest arg))
```

```
              (and (consp tmp1)
                   (progn
                    (setq v (qfirst tmp1))
                    (setq tmp2 (qrest tmp1))
                    (and (consp tmp2) (eq (qrest tmp2) nil)
                         (progn
                          (setq tt (qfirst tmp2))
                          t))))))
         (setq sig1 (cons tt sig1))
         (setq arg1 (cons v arg1)))
        (t (setq bad t))))
    (cond
     (bad (list nil nil ))
     (t (list (reverse arg1) (reverse sig1)))))))
```

———-

14.2.54 defun compMakeDeclaration

[compColon p271]
[$insideExpressionIfTrue p??]

— defun compMakeDeclaration —

```
(defun |compMakeDeclaration| (form mode env)
 (let (|$insideExpressionIfTrue|)
 (declare (special |$insideExpressionIfTrue|))
  (setq |$insideExpressionIfTrue| nil)
  (|compColon| form mode env)))
```

———-

14.2.55 defun modifyModeStack

[say p??]
[copy p??]
[setelt p??]
[resolve p334]
[$reportExitModeStack p??]
[$exitModeStack p??]

— defun modifyModeStack —

```
(defun |modifyModeStack| (m index)
 (declare (special |$exitModeStack| |$reportExitModeStack|))
 (if |$reportExitModeStack|
   (say "exitModeStack: " (copy |$exitModeStack|)
    " ====> "
    (progn
     (setelt |$exitModeStack| index
      (|resolve| m (elt |$exitModeStack| index)))
```

```
    |$exitModeStack|))
  (setelt |$exitModeStack| index
     (|resolve| m (elt |$exitModeStack| index)))))
```

———-

14.2.56 defun Create a list of unbound symbols

We walk argument u looking for symbols that are unbound. If we find a symbol we add it to the free list. If it occurs in a prog then it is bound and we remove it from the free list. Multiple instances of a single symbol in the free list are represented by the alist (symbol . count) [freelist p562]
[freelist assq (vol5)]
[freelist identp (vol5)]
[getmode p??]
[unionq p??]

— defun freelist —

```
(defun freelist (u bound free e)
 (let (v op)
  (if (atom u)
   (cond
    ((null (identp u)) free)
    ((member u bound) free)
    ; more than 1 free becomes alist (name . number)
    ((setq v (assq u free)) (rplacd v (+ 1 (cdr v))) free)
    ((null (|getmode| u e)) free)
    (t (cons (cons u 1) free)))
   (progn
    (setq op (car u))
    (cond
     ((member op '(quote go function)) free)
     ((eq op 'lambda) ; lambdas bind symbols
      (setq bound (unionq bound (second u)))
      (dolist (v (cddr u))
       (setq free (freelist v bound free e))))
     ((eq op 'prog) ; progs bind symbols
      (setq bound (unionq bound (second u)))
      (dolist (v (cddr u))
       (unless (atom v)
        (setq free (freelist v bound free e)))))
     ((eq op 'seq)
      (dolist (v (cdr u))
       (unless (atom v)
        (setq free (freelist v bound free e)))))
     ((eq op 'cond)
      (dolist (v (cdr u))
       (dolist (vv v)
        (setq free (freelist vv bound free e)))))
     (t
      (when (atom op) (setq u (cdr u))) ; atomic functions aren't descended
```

```
      (dolist (v u)
       (setq free (freelist v bound free e)))))
    free))))
```

———-

14.2.57 defun compOrCroak1,compactify

[compOrCroak1,compactify p563]
[lassoc p??]

— defun compOrCroak1,compactify —

```
(defun |compOrCroak1,compactify| (al)
 (cond
  ((null al) nil)
  ((lassoc (caar al) (cdr al)) (|compOrCroak1,compactify| (cdr al)))
  (t (cons (car al) (|compOrCroak1,compactify| (cdr al))))))
```

———-

14.2.58 defun Compiler/Interpreter interface

[ncINTERPFILE SpadInterpretStream (vol5)]
[$EchoLines p??]
[$ReadingFile p??]
The **SpadInterpretStream** function call takes three arguments. The first argument The second argument **source** is the name of a file to include. The third argument **interactive?**, when false, will read from the file rather than the console.

— defun ncINTERPFILE —

```
(defun |ncINTERPFILE| (file echo)
 (let ((|$EchoLines| echo) (|$ReadingFile| t))
 (declare (special |$EchoLines| |$ReadingFile|))
  (|SpadInterpretStream| 1 file nil)))
```

———-

14.2.59 defun recompile-lib-file-if-necessary

[compile-lib-file p564]
[*lisp-bin-filetype* p??]

— defun recompile-lib-file-if-necessary —

```
(defun recompile-lib-file-if-necessary (lfile)
 (let* ((bfile (make-pathname :type *lisp-bin-filetype* :defaults lfile))
        (bdate (and (probe-file bfile) (file-write-date bfile)))
        (ldate (and (probe-file lfile) (file-write-date lfile))))
  (declare (special *lisp-bin-filetype*))
```

```
  (unless (and ldate bdate (> bdate ldate))
   (compile-lib-file lfile)
   (list bfile))))
```

———-

14.2.60 defun spad-fixed-arg

— defun spad-fixed-arg —

```
(defun spad-fixed-arg (fname )
 (and (equal (symbol-package fname) (find-package "BOOT"))
      (not (get fname 'compiler::spad-var-arg))
      (search ";" (symbol-name fname))
      (or (get fname 'compiler::fixed-args)
          (setf (get fname 'compiler::fixed-args) t)))
   nil)
```

———-

14.2.61 defun compile-lib-file

— defun compile-lib-file —

```
(defun compile-lib-file (fn &rest opts)
 (unwind-protect
  (progn
   (trace (compiler::fast-link-proclaimed-type-p
           :exitcond nil
           :entrycond (spad-fixed-arg (car system::arglist))))
   (trace (compiler::t1defun
           :exitcond nil
           :entrycond (spad-fixed-arg (caar system::arglist))))
   (apply #'compile-file fn opts))
  (untrace compiler::fast-link-proclaimed-type-p compiler::t1defun)))
```

———-

14.2.62 defun compileFileQuietly

if `$InteractiveMode` then use a null outputstream [$InteractiveMode p??]
[*standard-output* p??]

— defun compileFileQuietly —

```
(defun |compileFileQuietly| (fn)
  (let (
        (*standard-output*
        (if |$InteractiveMode| (make-broadcast-stream)
```

```
          *standard-output*)))
  (declare (special *standard-output* |$InteractiveMode|))
  (compile-file fn)))
```

———-

14.2.63 defvar $byConstructors

— initvars —

```
(defvar |$byConstructors| () "list of constructors to be compiled")
```

———-

14.2.64 defvar $constructorsSeen

— initvars —

```
(defvar |$constructorsSeen| () "list of constructors found")
```

———-

Chapter 15

Level 1

15.0.65 defvar current-fragment

A string containing remaining chars from readline; needed because Symbolics read-line returns embedded newlines in a c-m-Y.

— **initvars** —

```
(defvar current-fragment nil)
```

———-

15.0.66 defun read-a-line

The **read-a-line** function reads a line from the current stream, potentially setting *eof*. It then recursively calls itself [read-a-line line-new-line (vol5)]
[read-a-line current-line (vol5)]
[*eof* p??]
[File-Closed p??]

read-a-line : **FileStream** → **String** where FileStream might be

```
#<input stream "/research/t1/src/algebra/EQ.spad">
```

and the returned string might be

```
")abbrev domain EQ Equation"
```

— **defun read-a-line** —

```
(defun read-a-line (&optional (stream t))
 (if (stream-eof stream)
  (progn
   (setq File-Closed t)
   (setq *eof* t)
   (line-new-line (make-string 0) current-line)
   nil)
  (read-line stream)))
```

Chapter 16

The Chunks

— Compiler —

```
(in-package "BOOT")

\getchunk{initvars}

\getchunk{LEDNUDTables}
\getchunk{GLIPHTable}
\getchunk{RENAMETOKTable}
\getchunk{GENERICTable}

\getchunk{defmacro bang}
\getchunk{defmacro must}
\getchunk{defmacro nth-stack}
\getchunk{defmacro pop-stack-1}
\getchunk{defmacro pop-stack-2}
\getchunk{defmacro pop-stack-3}
\getchunk{defmacro pop-stack-4}
\getchunk{defmacro reduce-stack-clear}
\getchunk{defmacro stack-/-empty}
\getchunk{defmacro star}

\getchunk{defun action}
\getchunk{defun addArgumentConditions}
\getchunk{defun addclose}
\getchunk{defun addConstructorModemaps}
\getchunk{defun addDomain}
\getchunk{defun addEltModemap}
\getchunk{defun addEmptyCapsuleIfNecessary}
\getchunk{defun addModemapKnown}
\getchunk{defun addModemap}
\getchunk{defun addModemap0}
\getchunk{defun addModemap1}
\getchunk{defun addNewDomain}
\getchunk{defun add-parens-and-semis-to-line}
\getchunk{defun addSuffix}
```

```
\getchunk{defun advance-token}
\getchunk{defun alistSize}
\getchunk{defun allLASSOCs}
\getchunk{defun aplTran}
\getchunk{defun aplTran1}
\getchunk{defun aplTranList}
\getchunk{defun applyMapping}
\getchunk{defun argsToSig}
\getchunk{defun assignError}
\getchunk{defun AssocBarGensym}
\getchunk{defun augLisplibModemapsFromCategory}
\getchunk{defun augmentLisplibModemapsFromFunctor}
\getchunk{defun augModemapsFromCategory}
\getchunk{defun augModemapsFromCategoryRep}
\getchunk{defun augModemapsFromDomain}
\getchunk{defun augModemapsFromDomain1}
\getchunk{defun autoCoerceByModemap}

\getchunk{defun blankp}
\getchunk{defun bootStrapError}
\getchunk{defun buildLibAttr}
\getchunk{defun buildLibAttrs}
\getchunk{defun buildLibdb}
\getchunk{defun buildLibdbConEntry}
\getchunk{defun buildLibdbString}
\getchunk{defun buildLibOp}
\getchunk{defun buildLibOps}
\getchunk{defun bumperrorcount}

\getchunk{defun canReturn}
\getchunk{defun char-eq}
\getchunk{defun char-ne}
\getchunk{defun checkAddBackSlashes}
\getchunk{defun checkAddIndented}
\getchunk{defun checkAddMacros}
\getchunk{defun checkAddPeriod}
\getchunk{defun checkAddSpaces}
\getchunk{defun checkAddSpaceSegments}
\getchunk{defun checkAlphabetic}
\getchunk{defun checkAndDeclare}
\getchunk{defun checkArguments}
\getchunk{defun checkBalance}
\getchunk{defun checkBeginEnd}
\getchunk{defun checkComments}
\getchunk{defun checkDecorate}
\getchunk{defun checkDecorateForHt}
\getchunk{defun checkDocError}
\getchunk{defun checkDocError1}
\getchunk{defun checkDocMessage}
\getchunk{defun checkExtract}
\getchunk{defun checkFixCommonProblem}
\getchunk{defun checkGetArgs}
\getchunk{defun checkGetLispFunctionName}
\getchunk{defun checkGetMargin}
```

```
\getchunk{defun checkGetParse}
\getchunk{defun checkGetStringBeforeRightBrace}
\getchunk{defun checkHTargs}
\getchunk{defun checkIeEg}
\getchunk{defun checkIeEgfun}
\getchunk{defun checkIndentedLines}
\getchunk{defun checkIsValidType}
\getchunk{defun checkLookForLeftBrace}
\getchunk{defun checkLookForRightBrace}
\getchunk{defun checkNumOfArgs}
\getchunk{defun checkRecordHash}
\getchunk{defun checkRemoveComments}
\getchunk{defun checkRewrite}
\getchunk{defun checkSayBracket}
\getchunk{defun checkSkipBlanks}
\getchunk{defun checkSkipIdentifierToken}
\getchunk{defun checkSkipOpToken}
\getchunk{defun checkSkipToken}
\getchunk{defun checkSplitBackslash}
\getchunk{defun checkSplitBrace}
\getchunk{defun checkSplitOn}
\getchunk{defun checkSplitPunctuation}
\getchunk{defun checkSplit2Words}
\getchunk{defun checkTexht}
\getchunk{defun checkTransformFirsts}
\getchunk{defun checkTrim}
\getchunk{defun checkTrimCommented}
\getchunk{defun checkWarning}
\getchunk{defun coerce}
\getchunk{defun coerceable}
\getchunk{defun coerceByModemap}
\getchunk{defun coerceEasy}
\getchunk{defun coerceExit}
\getchunk{defun coerceExtraHard}
\getchunk{defun coerceHard}
\getchunk{defun coerceSubset}
\getchunk{defun collectAndDeleteAssoc}
\getchunk{defun collectComBlock}
\getchunk{defun comma2Tuple}
\getchunk{defun comp}
\getchunk{defun comp2}
\getchunk{defun comp3}
\getchunk{defun compAdd}
\getchunk{defun compAndDefine}
\getchunk{defun compApplication}
\getchunk{defun compApply}
\getchunk{defun compApplyModemap}
\getchunk{defun compArgumentConditions}
\getchunk{defun compArgumentsAndTryAgain}
\getchunk{defun compAtom}
\getchunk{defun compAtomWithModemap}
\getchunk{defun compAtSign}
\getchunk{defun compBoolean}
\getchunk{defun compCapsule}
```

```
\getchunk{defun compCapsuleInner}
\getchunk{defun compCapsuleItems}
\getchunk{defun compCase}
\getchunk{defun compCase1}
\getchunk{defun compCat}
\getchunk{defun compCategory}
\getchunk{defun compCategoryItem}
\getchunk{defun compCoerce}
\getchunk{defun compCoerce1}
\getchunk{defun compColon}
\getchunk{defun compColonInside}
\getchunk{defun compCons}
\getchunk{defun compCons1}
\getchunk{defun compConstruct}
\getchunk{defun compConstructorCategory}
\getchunk{defun compDefine}
\getchunk{defun compDefine1}
\getchunk{defun compDefineAddSignature}
\getchunk{defun compDefineCapsuleFunction}
\getchunk{defun compDefineCategory}
\getchunk{defun compDefineCategory1}
\getchunk{defun compDefineCategory2}
\getchunk{defun compDefineFunctor}
\getchunk{defun compDefineFunctor1}
\getchunk{defun compDefineLisplib}
\getchunk{defun compDefWhereClause}
\getchunk{defun compElt}
\getchunk{defun compExit}
\getchunk{defun compExpression}
\getchunk{defun compExpressionList}
\getchunk{defun compForm}
\getchunk{defun compForm1}
\getchunk{defun compForm2}
\getchunk{defun compForm3}
\getchunk{defun compFormMatch}
\getchunk{defun compForMode}
\getchunk{defun compFormPartiallyBottomUp}
\getchunk{defun compFormWithModemap}
\getchunk{defun compFromIf}
\getchunk{defun compFunctorBody}
\getchunk{defun compHas}
\getchunk{defun compHasFormat}
\getchunk{defun compIf}
\getchunk{defun compile}
\getchunk{defun compileCases}
\getchunk{defun compileConstructor}
\getchunk{defun compileConstructor1}
\getchunk{defun compileDocumentation}
\getchunk{defun compileFileQuietly}
\getchunk{defun compile-lib-file}
\getchunk{defun compiler}
\getchunk{defun compilerDoit}
\getchunk{defun compilerDoitWithScreenedLisplib}
\getchunk{defun compileSpad2Cmd}
```

```
\getchunk{defun compileSpadLispCmd}
\getchunk{defun compileTimeBindingOf}
\getchunk{defun compImport}
\getchunk{defun compInternalFunction}
\getchunk{defun compIs}
\getchunk{defun compJoin}
\getchunk{defun compLambda}
\getchunk{defun compLeave}
\getchunk{defun compList}
\getchunk{defun compMacro}
\getchunk{defun compMakeCategoryObject}
\getchunk{defun compMakeDeclaration}
\getchunk{defun compMapCond}
\getchunk{defun compMapCond'}
\getchunk{defun compMapCond''}
\getchunk{defun compMapCondFun}
\getchunk{defun compNoStacking}
\getchunk{defun compNoStacking1}
\getchunk{defun compOrCroak}
\getchunk{defun compOrCroak1}
\getchunk{defun compOrCroak1,compactify}
\getchunk{defun compPretend}
\getchunk{defun compQuote}
\getchunk{defun compRepeatOrCollect}
\getchunk{defun compReduce}
\getchunk{defun compReduce1}
\getchunk{defun compReturn}
\getchunk{defun compSeq}
\getchunk{defun compSeqItem}
\getchunk{defun compSeq1}
\getchunk{defun compSetq}
\getchunk{defun compSetq1}
\getchunk{defun compSingleCapsuleItem}
\getchunk{defun compString}
\getchunk{defun compSubDomain}
\getchunk{defun compSubDomain1}
\getchunk{defun compSymbol}
\getchunk{defun compSubsetCategory}
\getchunk{defun compSuchthat}
\getchunk{defun compToApply}
\getchunk{defun compTopLevel}
\getchunk{defun compTuple2Record}
\getchunk{defun compTypeOf}
\getchunk{defun compUniquely}
\getchunk{defun compVector}
\getchunk{defun compWhere}
\getchunk{defun compWithMappingMode}
\getchunk{defun compWithMappingMode1}
\getchunk{defun constructMacro}
\getchunk{defun containsBang}
\getchunk{defun convert}
\getchunk{defun convertOpAlist2compilerInfo}
\getchunk{defun convertOrCroak}
\getchunk{defun current-char}
```

```
\getchunk{defun current-symbol}
\getchunk{defun current-token}

\getchunk{defun dbReadLines}
\getchunk{defun dbWriteLines}
\getchunk{defun decodeScripts}
\getchunk{defun deepestExpression}
\getchunk{defun def-rename}
\getchunk{defun disallowNilAttribute}
\getchunk{defun displayMissingFunctions}
\getchunk{defun displayPreCompilationErrors}
\getchunk{defun doIt}
\getchunk{defun doItIf}
\getchunk{defun dollarTran}
\getchunk{defun domainMember}
\getchunk{defun drop}

\getchunk{defun eltModemapFilter}
\getchunk{defun encodeItem}
\getchunk{defun encodeFunctionName}
\getchunk{defun EqualBarGensym}
\getchunk{defun escape-keywords}
\getchunk{defun escaped}
\getchunk{defun evalAndRwriteLispForm}
\getchunk{defun evalAndSub}
\getchunk{defun expand-tabs}
\getchunk{defun extendLocalLibdb}
\getchunk{defun extractCodeAndConstructTriple}

\getchunk{defun flattenSignatureList}
\getchunk{defun finalizeDocumentation}
\getchunk{defun finalizeLisplib}
\getchunk{defun fincomblock}
\getchunk{defun firstNonBlankPosition}
\getchunk{defun fixUpPredicate}
\getchunk{defun floatexpid}
\getchunk{defun formal2Pattern}
\getchunk{defun freelist}

\getchunk{defun getAbbreviation}
\getchunk{defun getArgumentMode}
\getchunk{defun getArgumentModeOrMoan}
\getchunk{defun getCaps}
\getchunk{defun getCategoryOpsAndAtts}
\getchunk{defun getConstructorExports}
\getchunk{defun getConstructorOpsAndAtts}
\getchunk{defun getDomainsInScope}
\getchunk{defun getFormModemaps}
\getchunk{defun getFunctorOpsAndAtts}
\getchunk{defun getInverseEnvironment}
\getchunk{defun getMatchingRightPren}
\getchunk{defun getModemap}
\getchunk{defun getModemapList}
\getchunk{defun getModemapListFromDomain}
```

```
\getchunk{defun getOperationAlist}
\getchunk{defun getScriptName}
\getchunk{defun getSignature}
\getchunk{defun getSignatureFromMode}
\getchunk{defun getSlotFromCategoryForm}
\getchunk{defun getSlotFromFunctor}
\getchunk{defun getSpecialCaseAssoc}
\getchunk{defun getSuccessEnvironment}
\getchunk{defun getTargetFromRhs}
\getchunk{defun get-token}
\getchunk{defun getToken}
\getchunk{defun getUnionMode}
\getchunk{defun getUniqueModemap}
\getchunk{defun getUniqueSignature}
\getchunk{defun genDomainOps}
\getchunk{defun genDomainViewList0}
\getchunk{defun genDomainViewList}
\getchunk{defun genDomainView}
\getchunk{defun giveFormalParametersValues}

\getchunk{defun hackforis}
\getchunk{defun hackforis1}
\getchunk{defun hasAplExtension}
\getchunk{defun hasFormalMapVariable}
\getchunk{defun hasFullSignature}
\getchunk{defun hasNoVowels}
\getchunk{defun hasSigInTargetCategory}
\getchunk{defun hasType}
\getchunk{defun htcharPosition}

\getchunk{defun indent-pos}
\getchunk{defun infixtok}
\getchunk{defun initialize-preparse}
\getchunk{defun initial-substring-p}
\getchunk{defun initializeLisplib}
\getchunk{defun insertModemap}
\getchunk{defun interactiveModemapForm}
\getchunk{defun isCategoryPackageName}
\getchunk{defun is-console}
\getchunk{defun isDomainConstructorForm}
\getchunk{defun isDomainForm}
\getchunk{defun isDomainSubst}
\getchunk{defun isFunctor}
\getchunk{defun isListConstructor}
\getchunk{defun isMacro}
\getchunk{defun isSuperDomain}
\getchunk{defun isTokenDelimiter}
\getchunk{defun isUnionMode}

\getchunk{defun killColons}

\getchunk{defun lispize}
\getchunk{defun lisplibDoRename}
\getchunk{defun lisplibWrite}
```

```
\getchunk{defun loadLibIfNecessary}

\getchunk{defun macroExpand}
\getchunk{defun macroExpandInPlace}
\getchunk{defun macroExpandList}
\getchunk{defun makeCategoryForm}
\getchunk{defun makeCategoryPredicates}
\getchunk{defun makeFunctorArgumentParameters}
\getchunk{defun makeSimplePredicateOrNil}
\getchunk{defun make-symbol-of}
\getchunk{defun match-advance-string}
\getchunk{defun match-current-token}
\getchunk{defun match-next-token}
\getchunk{defun match-string}
\getchunk{defun match-token}
\getchunk{defun maxSuperType}
\getchunk{defun mergeModemap}
\getchunk{defun mergeSignatureAndLocalVarAlists}
\getchunk{defun meta-syntax-error}
\getchunk{defun mkAbbrev}
\getchunk{defun mkAlistOfExplicitCategoryOps}
\getchunk{defun mkCategoryPackage}
\getchunk{defun mkConstructor}
\getchunk{defun mkDatabasePred}
\getchunk{defun mkEvalableCategoryForm}
\getchunk{defun mkExplicitCategoryFunction}
\getchunk{defun mkList}
\getchunk{defun mkNewModemapList}
\getchunk{defun mkOpVec}
\getchunk{defun mkRepititionAssoc}
\getchunk{defun mkUnion}
\getchunk{defun modifyModeStack}
\getchunk{defun modeEqual}
\getchunk{defun modeEqualSubst}
\getchunk{defun modemapPattern}
\getchunk{defun moveORsOutside}
\getchunk{defun mustInstantiate}

\getchunk{defun ncINTERPFILE}
\getchunk{defun newWordFrom}
\getchunk{defun next-char}
\getchunk{defun next-tab-loc}
\getchunk{defun next-token}
\getchunk{defun newConstruct}
\getchunk{defun newDef2Def}
\getchunk{defun newIf2Cond}
\getchunk{defun newString2Words}
\getchunk{defun new2OldDefForm}
\getchunk{defun new2OldTran}
\getchunk{defun new2OldLisp}
\getchunk{defun nonblankloc}
\getchunk{defun NRTassocIndex}
\getchunk{defun NRTgetLocalIndex}
\getchunk{defun NRTgetLookupFunction}
```

```
\getchunk{defun NRTputInHead}
\getchunk{defun NRTputInTail}

\getchunk{defun optCall}
\getchunk{defun optCallEval}
\getchunk{defun optCallSpecially}
\getchunk{defun optCatch}
\getchunk{defun optCond}
\getchunk{defun optCONDtail}
\getchunk{defun optEQ}
\getchunk{defun optIF2COND}
\getchunk{defun optimize}
\getchunk{defun optimizeFunctionDef}
\getchunk{defun optional}
\getchunk{defun optLESSP}
\getchunk{defun optMINUS}
\getchunk{defun optMkRecord}
\getchunk{defun optPackageCall}
\getchunk{defun optPredicateIfTrue}
\getchunk{defun optQSMINUS}
\getchunk{defun optRECORDCOPY}
\getchunk{defun optRECORDELT}
\getchunk{defun optSETRECORDELT}
\getchunk{defun optSEQ}
\getchunk{defun optSPADCALL}
\getchunk{defun optSpecialCall}
\getchunk{defun optSuchthat}
\getchunk{defun optXLAMCond}
\getchunk{defun opt-}
\getchunk{defun orderByDependency}
\getchunk{defun orderPredicateItems}
\getchunk{defun orderPredTran}
\getchunk{defun outputComp}

\getchunk{defun PARSE-AnyId}
\getchunk{defun PARSE-Application}
\getchunk{defun parse-argument-designator}
\getchunk{defun parse-identifier}
\getchunk{defun parse-keyword}
\getchunk{defun parse-number}
\getchunk{defun parse-spadstring}
\getchunk{defun parse-string}
\getchunk{defun PARSE-Category}
\getchunk{defun PARSE-Command}
\getchunk{defun PARSE-CommandTail}
\getchunk{defun PARSE-Conditional}
\getchunk{defun PARSE-Data}
\getchunk{defun PARSE-ElseClause}
\getchunk{defun PARSE-Enclosure}
\getchunk{defun PARSE-Exit}
\getchunk{defun PARSE-Expr}
\getchunk{defun PARSE-Expression}
\getchunk{defun PARSE-Float}
\getchunk{defun PARSE-FloatBase}
```

```
\getchunk{defun PARSE-FloatBasePart}
\getchunk{defun PARSE-FloatExponent}
\getchunk{defun PARSE-FloatTok}
\getchunk{defun PARSE-Form}
\getchunk{defun PARSE-FormalParameter}
\getchunk{defun PARSE-FormalParameterTok}
\getchunk{defun PARSE-getSemanticForm}
\getchunk{defun PARSE-GliphTok}
\getchunk{defun PARSE-Import}
\getchunk{defun PARSE-Infix}
\getchunk{defun PARSE-InfixWith}
\getchunk{defun PARSE-IntegerTok}
\getchunk{defun PARSE-Iterator}
\getchunk{defun PARSE-IteratorTail}
\getchunk{defun PARSE-Label}
\getchunk{defun PARSE-LabelExpr}
\getchunk{defun PARSE-Leave}
\getchunk{defun PARSE-LedPart}
\getchunk{defun PARSE-leftBindingPowerOf}
\getchunk{defun PARSE-Loop}
\getchunk{defun PARSE-Name}
\getchunk{defun PARSE-NBGliphTok}
\getchunk{defun PARSE-NewExpr}
\getchunk{defun PARSE-NudPart}
\getchunk{defun PARSE-OpenBrace}
\getchunk{defun PARSE-OpenBracket}
\getchunk{defun PARSE-Operation}
\getchunk{defun PARSE-Option}
\getchunk{defun PARSE-Prefix}
\getchunk{defun PARSE-Primary}
\getchunk{defun PARSE-Primary1}
\getchunk{defun PARSE-PrimaryNoFloat}
\getchunk{defun PARSE-PrimaryOrQM}
\getchunk{defun PARSE-Qualification}
\getchunk{defun PARSE-Quad}
\getchunk{defun PARSE-Reduction}
\getchunk{defun PARSE-ReductionOp}
\getchunk{defun PARSE-Return}
\getchunk{defun PARSE-rightBindingPowerOf}
\getchunk{defun PARSE-ScriptItem}
\getchunk{defun PARSE-Scripts}
\getchunk{defun PARSE-Seg}
\getchunk{defun PARSE-Selector}
\getchunk{defun PARSE-SemiColon}
\getchunk{defun PARSE-Sequence}
\getchunk{defun PARSE-Sequence1}
\getchunk{defun PARSE-Sexpr}
\getchunk{defun PARSE-Sexpr1}
\getchunk{defun PARSE-SpecialCommand}
\getchunk{defun PARSE-SpecialKeyWord}
\getchunk{defun PARSE-Statement}
\getchunk{defun PARSE-String}
\getchunk{defun PARSE-Suffix}
\getchunk{defun PARSE-TokenCommandTail}
```

```
\getchunk{defun PARSE-TokenList}
\getchunk{defun PARSE-TokenOption}
\getchunk{defun PARSE-TokTail}
\getchunk{defun PARSE-VarForm}
\getchunk{defun PARSE-With}
\getchunk{defun parsepiles}
\getchunk{defun parseAnd}
\getchunk{defun parseAtom}
\getchunk{defun parseAtSign}
\getchunk{defun parseCategory}
\getchunk{defun parseCoerce}
\getchunk{defun parseColon}
\getchunk{defun parseConstruct}
\getchunk{defun parseDEF}
\getchunk{defun parseDollarGreaterEqual}
\getchunk{defun parseDollarGreaterThan}
\getchunk{defun parseDollarLessEqual}
\getchunk{defun parseDollarNotEqual}
\getchunk{defun parseDropAssertions}
\getchunk{defun parseEquivalence}
\getchunk{defun parseExit}
\getchunk{defun postFlatten}
\getchunk{defun postFlattenLeft}
\getchunk{defun postForm}
\getchunk{defun parseGreaterEqual}
\getchunk{defun parseGreaterThan}
\getchunk{defun parseHas}
\getchunk{defun parseHasRhs}
\getchunk{defun parseIf}
\getchunk{defun parseIf,ifTran}
\getchunk{defun parseImplies}
\getchunk{defun parseIn}
\getchunk{defun parseInBy}
\getchunk{defun parseIs}
\getchunk{defun parseIsnt}
\getchunk{defun parseJoin}
\getchunk{defun parseLeave}
\getchunk{defun parseLessEqual}
\getchunk{defun parseLET}
\getchunk{defun parseLETD}
\getchunk{defun parseLhs}
\getchunk{defun parseMDEF}
\getchunk{defun parseNot}
\getchunk{defun parseNotEqual}
\getchunk{defun parseOr}
\getchunk{defun parsePretend}
\getchunk{defun parseprint}
\getchunk{defun parseReturn}
\getchunk{defun parseSegment}
\getchunk{defun parseSeq}
\getchunk{defun parseTran}
\getchunk{defun parseTranCheckForRecord}
\getchunk{defun parseTranList}
\getchunk{defun parseTransform}
```

```
\getchunk{defun parseType}
\getchunk{defun parseVCONS}
\getchunk{defun parseWhere}
\getchunk{defun Pop-Reduction}
\getchunk{defun postAdd}
\getchunk{defun postAtom}
\getchunk{defun postAtSign}
\getchunk{defun postBigFloat}
\getchunk{defun postBlock}
\getchunk{defun postBlockItem}
\getchunk{defun postBlockItemList}
\getchunk{defun postCapsule}
\getchunk{defun postCategory}
\getchunk{defun postcheck}
\getchunk{defun postCollect}
\getchunk{defun postCollect,finish}
\getchunk{defun postColon}
\getchunk{defun postColonColon}
\getchunk{defun postComma}
\getchunk{defun postConstruct}
\getchunk{defun postDef}
\getchunk{defun postDefArgs}
\getchunk{defun postError}
\getchunk{defun postExit}
\getchunk{defun postIf}
\getchunk{defun postin}
\getchunk{defun postIn}
\getchunk{defun postInSeq}
\getchunk{defun postIteratorList}
\getchunk{defun postJoin}
\getchunk{defun postMakeCons}
\getchunk{defun postMapping}
\getchunk{defun postMDef}
\getchunk{defun postOp}
\getchunk{defun postPretend}
\getchunk{defun postQUOTE}
\getchunk{defun postReduce}
\getchunk{defun postRepeat}
\getchunk{defun postScripts}
\getchunk{defun postScriptsForm}
\getchunk{defun postSemiColon}
\getchunk{defun postSignature}
\getchunk{defun postSlash}
\getchunk{defun postTran}
\getchunk{defun postTranList}
\getchunk{defun postTranScripts}
\getchunk{defun postTranSegment}
\getchunk{defun postTransform}
\getchunk{defun postTransformCheck}
\getchunk{defun postTuple}
\getchunk{defun postTupleCollect}
\getchunk{defun postType}
\getchunk{defun postWhere}
\getchunk{defun postWith}
```

```
\getchunk{defun preparse}
\getchunk{defun preparse1}
\getchunk{defun preparse-echo}
\getchunk{defun preparseReadLine}
\getchunk{defun preparseReadLine1}
\getchunk{defun primitiveType}
\getchunk{defun print-defun}
\getchunk{defun processFunctor}
\getchunk{defun purgeNewConstructorLines}
\getchunk{defun push-reduction}
\getchunk{defun putDomainsInScope}
\getchunk{defun putInLocalDomainReferences}

\getchunk{defun quote-if-string}

\getchunk{defun read-a-line}
\getchunk{defun recompile-lib-file-if-necessary}
\getchunk{defun recordAttributeDocumentation}
\getchunk{defun recordDocumentation}
\getchunk{defun recordHeaderDocumentation}
\getchunk{defun recordSignatureDocumentation}
\getchunk{defun replaceExitEtc}
\getchunk{defun removeBackslashes}
\getchunk{defun removeSuperfluousMapping}
\getchunk{defun replaceVars}
\getchunk{defun resolve}
\getchunk{defun reportOnFunctorCompilation}
\getchunk{defun /rf}
\getchunk{defun /rq}
\getchunk{defun /rf-1}
\getchunk{defun /RQ,LIB}
\getchunk{defun rwriteLispForm}

\getchunk{defun screenLocalLine}
\getchunk{defun setDefOp}
\getchunk{defun seteltModemapFilter}
\getchunk{defun setqMultiple}
\getchunk{defun setqMultipleExplicit}
\getchunk{defun setqSetelt}
\getchunk{defun setqSingle}
\getchunk{defun signatureTran}
\getchunk{defun skip-blanks}
\getchunk{defun skip-ifblock}
\getchunk{defun skip-to-endif}
\getchunk{defun spad}
\getchunk{defun spadCompileOrSetq}
\getchunk{defun spad-fixed-arg}
\getchunk{defun spadSysBranch}
\getchunk{defun spadSysChoose}
\getchunk{defun splitEncodedFunctionName}
\getchunk{defun stack-clear}
\getchunk{defun stack-load}
\getchunk{defun stack-pop}
\getchunk{defun stack-push}
```

```
\getchunk{defun string2BootTree}
\getchunk{defun stripOffArgumentConditions}
\getchunk{defun stripOffSubdomainConditions}
\getchunk{defun subrname}
\getchunk{defun substituteCategoryArguments}
\getchunk{defun substituteIntoFunctorModemap}
\getchunk{defun substNames}
\getchunk{defun substVars}
\getchunk{defun s-process}

\getchunk{defun token-install}
\getchunk{defun token-lookahead-type}
\getchunk{defun token-print}
\getchunk{defun transDoc}
\getchunk{defun transDocList}
\getchunk{defun transformAndRecheckComments}
\getchunk{defun transformOperationAlist}
\getchunk{defun transImplementation}
\getchunk{defun transIs}
\getchunk{defun transIs1}
\getchunk{defun translabel}
\getchunk{defun translabel1}
\getchunk{defun TruthP}
\getchunk{defun try-get-token}
\getchunk{defun tuple2List}

\getchunk{defun uncons}
\getchunk{defun underscore}
\getchunk{defun unget-tokens}
\getchunk{defun unknownTypeError}
\getchunk{defun unloadOneConstructor}
\getchunk{defun unTuple}
\getchunk{defun updateCategoryFrameForCategory}
\getchunk{defun updateCategoryFrameForConstructor}

\getchunk{defun whoOwns}
\getchunk{defun wrapDomainSub}
\getchunk{defun writeLib1}

\getchunk{postvars}
```

Bibliography

[Bott17] Gunther Rademacher. Railroad Diagram Generator, 2017.

Link: http://www.bottlecaps.de/rr/ui

[Dave84a] James H. Davenport. A New Algebra System.

Abstract: Seminal internal paper discussing Axiom design decisions.

Link: http://axiom-wiki.newsynthesis.org/public/refs/Davenport-1984-a_new_algebra_system.pdf

[Dosr11] Gabriel Dos Reis, David Matthews, and Yue Li. *Retargeting OpenAxiom to Poly/ML: Towards an Integrated Proof Assistants and Computer Algebra System Framework*, pages 15–29. Springer, 2011, 978-3-642-22673-1.

Abstract: This paper presents an ongoing effort to integrate the Axiom family of computer algebra systems with Poly/ML-based proof assistants in the same framework. A long term goal is to make a large set of efficient implementations of algebraic algorithms available to popular proof assistants, and also to bring the power of mechanized formal verification to a family of strongly typed computer algebra systems at a modest cost. Our approach is based on retargeting the code generator of the OpenAxiom compiler to the Poly/ML abstract machine.

Link: http://paradise.caltech.edu/~yli/paper/oa-polyml.pdf

[Floy63] R. W. Floyd. Semantic Analysis and Operator Precedence. *JACM*, 10(3):316–333, 1963.

[Knut92] Donald E. Knuth. *Literate Programming.* Center for the Study of Language and Information, Stanford CA, 1992, 0-937073-81-4.

[Prat73] Vaughan R. Pratt. Top down operator precedence. In *Proc. 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL'73, pages 41–51, 1973.

Link: http://hall.org.ua/halls/wizzard/pdf/Vaughan.Pratt.TDOP.pdf

[Smit10] Jacob Nyffeler Smith. *Techniques in Active and Generic Software Libraries.* PhD thesis, Texas A and M University, 2010.

Abstract: Reusing code from software libraries can reduce the time and effort to construct software systems and also enable the development of larger systems. However, the benefits that come from the use of software libraries may not be realized due to limitations in the way that traditional software libraries are constructed. Libraries come equipped with application programming interfaces (API) that help enforce the correct use of the abstraction in those libraries. Writing new components and adapting existing ones to conform to library APIs may require substantial amounts of "glue" code that potentially affects software's efficiency, robustness, and ease-of-maintenance. If, as a result, the idea of reusing functionality from a software library is rejected, no benefits of reuse will be realized. This dissertation explores and develops techniques that support the construction of software libraries with abstraction layers that do not impede efficiency. In many situations, glue code can be expected to have very low (or zero) performance overhead. In particular, we describe advances in the design and development of active libraries – software libraries that take an active role in the compilation of the user's code. Common to the presented techniques is that they may "break" a library API (in a controlled manner) to adapt the functionality of the library for a particular use case. The concrete contributions of this dissertation are: a library API that supports iterator selection in the Standard Template Library, allowing generic algorithms to find the most suitable traversal through a container, allowing (in one case) a 30-fold improvement in performance; the development of techniques, idioms, and best practices for `concepts` and `concept_maps` in C++, allowing the construction of algorithms for one domain entirely in terms of formalisms from a second domain; the construction of generic algorithms for algorithmic differentiation, implemented as an active library in Spad, language of the Open Axiom computer algebra system, allowing algorithmic differentiation to be applied to the appropriate mathematical object and not just concrete data-types; and the description of a static analysis framework to describe the generic programming notion of local specialization with Spad, allowing more sophisticated (value-based) control over algorithm selection and specialization in categories and domains. We will find that active libraries simultaneously increase the expressivity of the underlying language and the performance of software using those libraries

Link: oaktrust.library.tamu.edu/bitstream/handle/1969.1/ETD-TAMU-2010-05-7823/SMITH-DISSERTATION.pdf

Index