The Weyl Computer Algebra Substrate Manual

Richard Zippel[1]
Cornell University
Ithaca, NY 14853
rz@cs.cornell.edu

November 15, 1991

[1]This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014–88–K–0591, the National Science Foundation through grant IRI–9006137, the Office of Naval Research through contract N00014–89–J–1946 and in part by the U.S. Army Research Office through the Mathematical Science Institute of Cornell University.

Contents

Chapter 1

Introduction

In the last twenty years the algorithms and techniques for manipulating symbolic mathematical quantities have improved dramatically. These techniques have been made available to a practitioners through a number of algebraic manipulation systems. Among the most widely distributed systems are Macsyma [9], Reduce [3], Maple [5] and Mathematica [10]. These systems are designed to be self-contained and are not intended to be incorporated into larger, more specialized systems. This is at odds with the experience with numerical computation where libraries of carefully coded routines like Linpack [2] and Eispack [7] have been of the greatest value because they could be incorporated in larger systems (like fluid dynamics simulators or circuit analyzers). Linear algebra systems like Matlab [6], though of significant value, where developed much later and are tend to be used more for research in linear algebra than as part of the large computations.

Another limitation of the symbolic mathematics sytems is that they deal with a relatively limited and fixed set of algebraic types which the user is not expected to extend significantly. Thus it may be difficult for a user to experiment with algorithms for dealing with Poisson series if the algebra system does not already have a data type that matches the behavior of Poisson series. This situation is exacerbated by the unavailability of the code that implements the algebraic algorithms of the symbolic manipulation systems. Scratchpad [4] is a noticable exception to this trend in that the developers plan to make the algebra code for the system widely available and the internal data typing mechanisms provided by Scratchpad are designed with the extension mechanisms just mentioned in mind.

Weyl is an extensible algebraic manipulation substrate that has been designed to represent all types of algebraic objects. It deals not only with the basic symbolic objects like polynomials, algebraic functions and differential forms, but can also deal with higher level objects like groups, rings, ideals and vector spaces. Furthermore, to encourage the use of symbolic techniques within other applications, Weyl is implemented as an extension of Common Lisp [8] using the Common Lisp Object Standard [1] so that all of Common Lisp's facilities and development tools can be used in concert with Weyl's symbolic tools.

It should be noted that the initial implementation of Weyl is intended to be as clean and semantically correct as possible. The primary goal of Weyl is provide the tools needed to express algebraic algorithms naturally and succintly. Nonetheless, we believe that algebraic algorithms can be efficiently implemented within the Weyl framework even though that was not a goal of the initial implementation.

Chapter 2

Getting Started

Weyl is implemented as a large body of functions that extend Lisp's functionality to include non-numeric algebraic computations. These functions can be loaded into an existing Lisp image using the tools described in Section 2.1. All of Weyl's extensions are attached to symbols in the `weyl` or `weyli` packages. The philosophy behind these packages and how they are to be used is discussed in Section 2.2. Some issues with developing programs using Weyl are discussed in Section 2.3. In Section 2.4 we give a simple of example what is involved when using Weyl.

2.1 Creating a Weyl World

The first thing one must do to get Weyl started is to load the file `sysdef.lisp`. This file creates the `weyl` package, where all of Weyl's symbols are kept, describes which files comprise the Weyl systems and provides a number utility functions to help manage the Weyl sources. These functions are all defined in the `user` package, which is the package in which Lisp's usually start up. Thus you will probably not need to include the `user::` prefix which we have included in this section.

The function `user::compile-weyl`, will recompile any Weyl sources that need recompilation. To load the the recompiled files use `user::load-weyl`.

`user::compile-weyl`

Compile those Weyl source files that need recompilation. Increment Weyl's minor version number.

`user::load-weyl`

Load all binary files for Weyl and initialize Weyl's data structures. If Weyl has already been loaded, this function loads any binary fiels that are more recent than the ones already loaded. This is commonly used as a stripped down "`load-patches`."

`user::dump-weyl`

Load's the current binary files for Weyl and dumps a world.

The version number of Weyl is maintained in the file `Weyl-Version`. The version consists of two components, a major and minor version number. The minor version number is updated each time `compile-weyl` is called. The major version number is only updated by explicit calls to `next-major-weyl-version`.

`user::next-major-weyl-version`
Update the major version of Weyl.

Since Common Lisp does not include a `defsystem` facility, Weyl includes a modification of the one developed by Doug Rand of Prime, Inc. Some documentation for this system is provided in Chapter 11.

2.2 Packages

In Common Lisp values and variables are attached to symbols. Collections of symbols are organized into *packages* to control access to the values and functions. The symbols used by Weyl are contained in two Common Lisp packages, `weyl` and `weyl-internals`. The `weyl-internals` package has a nickname of `weyli`, which can be used interchangably with `weyl-internals`. Those symbols that are intended for users are contained in the `weyl` package and exported. The internal routines of Weyl are contained in the `weyl-internals` package so that they will not interfer with the user's symbols.

Common Lisp places all of its own symbols in the `lisp` package. Weyl extends the functionality of certain function in Common Lisp by providing its own version of certain symbols from the `lisp` package. Weyl's version of these symbols *shadow* the Common Lisp symbols. Among these symbols are the arithemetic functions, (`+`, `-`, `*`, ...), most of the mathematical functions (`sqrt`, `sin`, ...) and some of the sequence functions (`map`, `delete`, `member`, ...). The variable `*weyli-shadowed-symbols*` contains a list of all shadowed symbols. In all cases, when applied to arguments admissable to the Common Lisp version of the function, the Weyl versions of these functions perform in exactly the same fashion as the Common Lisp version. The only difference is that the Weyl versions have been extended to work with the mathematical objected defined in Weyl.

2.2.1 Difference between `weyl` and `weyli`

The only symbols in the `weyl` package are those that are intended for users. The internal functions and values that are maintained by the system itself and that are needed only by those extending Weyl are contained in the `weyl-internals` package.

In some cases are the are symbols with the same name in both the `weyl` and the `weyl-internals` packages. This is usually done when a version with more error checking is needed by users, but where such error checking would compromise the performance of Weyl if used internally where such error checking is not necessary.

An example of this is the `make-element` function. When constructing vectors and matrices, `weyl:make-element` makes sure that the each element of the vector or matrix to be created belongs to the coefficient domain, and if not tries to coerce the element into the coefficient domain. The function `weyli::make-element` does not, but it is only used internally in the vector and matrix routines.

2.2.2 Creating your own package

Most projects that use Weyl will create their own package for their routines. Because of the shadowing done by the weyl package, if one simply included the `weyl` in the user's new package one would get many conflicts. Instead one should use the function `use-weyl-package` which will perform the necessary shadowing.

`use-weyl-package` *package*

> Does a shadowing import of the shadowed symbols in the `weyl` package and then makes *package* use the package `weyl`.

Thus a common idiom for creating a package for an new application thst uses Weyl is

```
(unless (find-package 'Application-Package)
  (make-package "Application-Package" :use '(lisp clos))
  (use-weyl-package (find-package 'Application-Package)))
```

2.3 Programming Style

In many cases the symbols in the `weyl` package shadow normal Common Lisp symbols. When this is done, the functions in the `weyl` package are upward compatible extensions of the Common Lisp ones. For instance, the Common Lisp function `lisp:expt`, can only be applied to Lisp numbers, while the function `weyl:expt` can be applied to polynomials, rational functions and other mathematical objects as well as numbers. In all cases, the `weyl` versions of these shadowed functions are upwardly compatible with the Common Lisp versions.

We have adopted the convention that the name of all boolean predicates end with a `?`. This is different from the convention adopted by Common Lisp, where most predicates end with a `p`. Thus the Weyl function which determines if a number is positive is `plus?`, where the Common Lisp function is `plusp`.

2.4 Example 1

In this section we show how to perform a simple algebraic calculation using Weyl. Throughout this section we will use terms, such as *domains*, that are not yet well defined. These terms will be defined in later sections.

First one needs to start a Lisp image that contains Weyl. Using Lucid's lisp on Decstation we get a prompt like:

```
;;; Lucid Common Lisp/DECsystem
;;; Development Environment Version 4.0, 12 November 1990
;;; Copyright (C) 1985, 1986, 1987, 1988, 1989, 1990 by Lucid, Inc.
;;; All Rights Reserved
;;;
;;; This software product contains confidential and trade secret information
;;; belonging to Lucid, Inc.  It may not be copied for any reason other than
;;; for archival and backup purposes.
```

```
;;;
;;; Lucid Common Lisp is a trademark of Lucid, Inc.
;;; DECsystem is a trademark of Digital Equipment Corp.

;;; Weyl Version 4.55, saved 14:38 Thursday, October 3, 1991

>
```

The first thing that should be done at this point is to type `(user::load-weyl)` which will cause any bug fixes and improvements that have been created since the Lisp image was dumped, as illustrated below:

```
> (user::load-weyl)
;;; Loading file #P"/amd/nori/a/weyl/develop/lisp-support.mbin"
;;; Done loading system USER::WEYL
;;; Weyl 4.56 loaded.

> (in-package 'weyl)
#<Package "WEYL" 1028943E>
```

We have also switched the `weyl` package so that can easily access all the functions in Weyl. We are now ready to begin.

Computations using Weyl are usually involve first creating a set of domains in which to perform a computation, creating a few elements of these domains to initiate the computation and then the calculation itself. This is illustrated in the following routines that computes the coefficients of the "F and G series." These coefficients are defined by the following recurrences.

$$\begin{aligned} f_0 &= 0 \\ f_n &= -\mu g_{n-1} - \sigma(\mu + 2\epsilon)\frac{\partial f_{n-1}}{\partial \epsilon} + (\epsilon - 2\sigma^2)\frac{\partial f_{n-1}}{\partial \sigma} - 3\mu\sigma\frac{\partial f_{n-1}}{\partial \mu} \\ g_0 &= 1 \\ g_n &= f_{n-1} - \sigma(\mu + 2\epsilon)\frac{\partial g_{n-1}}{\partial \epsilon} + (\epsilon - 2\sigma^2)\frac{\partial g_{n-1}}{\partial \sigma} - 3\mu\sigma\frac{\partial g_{n-1}}{\partial \mu} \end{aligned}$$

The resulting sequences, f_i and g_i are all polynomials in μ, ϵ and σ with integer coefficients. Thus they lie in the domain $\mathbb{Z}[\mu, \epsilon, \sigma]$. Thus our first task in computing them is to generate the domain $\mathbb{Z}[\mu, \epsilon, \sigma]$. This can be done by the following line:

```
> (setf R (get-polynomial-ring (get-rational-integers) '(m e s)))
Z[m, e, s]
```

It is useful to have our hands on the variables μ, ϵ and σ. This is done by coercing the symbols `m`, `e` and `s` into the domain `R`.

```
> (setf mu (coerce 'm R))
m
> (setf eps (coerce 'e R))
e
```

```
> (setf sigma (coerce 's R))
s
```

The coefficients of the first terms of the both recurrences are the same, so rather than computing them afresh each time, we'll store them in some global variables

```
> (setq x1 (* (- sigma) (+ mu (* 2 eps))))
-1 s m + -2 s e

> (setq x2 (+ eps (* -2 (expt sigma 2))))
e + -2 s^2

> (setq x3 (* -3 mu sigma))
-3 s m

> (setq x4 (- mu))
 -1 m
```

Notice that we used the usual Lisp functions for manipulating the elements of `R`. Second, there is an automatic coercion from the integers into almost every domain we are working with, thus we could write `(* 2 eps)` above, rather than the more explicit `(* (coerce 2 R) eps)`.

Finally, we can write out the recursion formulae.

```
(defun f (n)
  (if (= n 0) (coerce 0 R)
      (+ (* x4 (g (1- n)))
         (* x1 (partial-deriv (f (1- n)) eps))
         (* x2 (partial-deriv (f (1- n)) sigma))
         (* x3 (partial-deriv (f (1- n)) mu)))))

(defun g (n)
  (if (= n 0) (coerce 1 R)
      (+ (f (1- n))
         (* x1 (partial-deriv (g (1- n)) eps))
         (* x2 (partial-deriv (g (1- n)) sigma))
         (* x3 (partial-deriv (g (1- n)) mu)))))
```

And compute f_5 and g_5

```
> (f 5)
-1 m^3 + (-24 e + 210 s^2) m^2 + (-45 e^2 + 630 s^2 e + -945 s^4) m

> (g 5)
-30 s m^2 + (-180 s e + 420 s^3) m
```

Chapter 3

Basics

The objects manipulated by Weyl are *domains*, *domain elements* and *morphisms* between domains. In Section 3.1 we discuss *domains* and *domain elements*. Section 3.2 discusses morphisms that are used to define the relationships between elements of two domains and in Section 3.3 we illustrate how make use of these relationships as well as the default ones. Section 3.4 presents the hierarchy of domains that are being used in Weyl.

3.1 Domains

Examples of domains are the rational integers ($\mathbb{Z}$), polynomial rings ($\mathbb{Q}[x, y]$, $\mathbb{Z}_p[t]$) and vector spaces ($\mathbb{R}^3$, $P^3(\mathbb{Z})$). The elements of these domains are domain elements. Each domain element is the element of a single domain and this domain can be determined at run time from the element.

Both domains and domain elements are implemented as instances of CLOS classes. These classes are part of a conventional, multiple inheritance class hierarchy. The class hierarchy used when constructing domains includes classes corresponding to groups, rings, fields and many other familiar types of algebraic domains.

This approach allows us to provide information about domains that is often difficult to indicate if we could only specify information about the type of the domain's elements. For instance, in addition to the class `Ring` we also provide classes like `integral-domain`.[1] Whether a domain is an integral domain or just a ring does not change how the domain's elements are represented or the operations that can be performed on them. What it may affect is the algorithms that are used to implement the operations. In a less mathematical context, one might consider the difference between binary trees and balanced binary trees.

In addition we provide information about the permissible operations involving elements of the domain. For instance, an `abelian-group` must have a `plus` operation that can be applied to pairs of its elements. The result will be an element of the abelian group. This parallels the mathematical definition of a group where a group is a pair $(G, \times)$ consisting of a set of elements (G) and a binary operation ($\times$) that maps pairs of elements of G into G. Contrast this to the typical definition of a type as a set of objects that obeys a predicate.

[1] An integral domain R is ring that does not have any zero divisors. That is, for all $a, b \in R$, if $ab = 0$ then either a or b is zero.

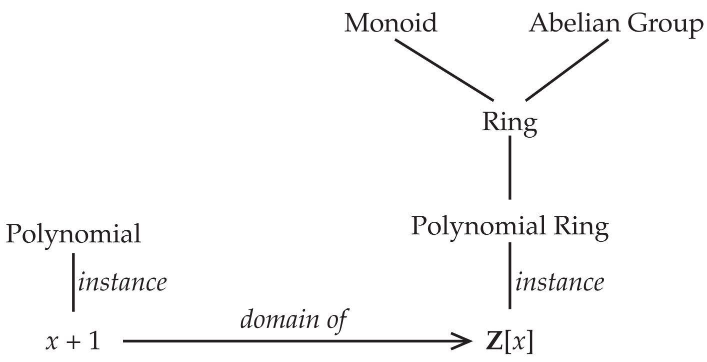

Figure 3.1: Domain Example

We attach information about the permissible operations on the elements of a domain to the domain itself. This allows us to ensure that when an instance of a domain is instantiated, the appropriate operations have been provided. This is especially valuable in an incremental development environment like those common for Lisp.

Domain elements are also implemented as instances of CLOS classes, but in addition they are associated with a particular domain. This association is implemented by a reference to the containing domain. The CLOS class structure provides the "structural type" referred to earlier.

Figure 3.1 illustrates the relationship between domains, domain elements and CLOS classes for a simple example. The polynomial $x + 1_{\mathbb{Z}[x]}$ is an instance of the CLOS class `polynomial`. The domain $\mathbb{Z}[x]$ is an instance of the class `polynomial-ring`, which inherits from `Ring`, `monoid` and `abelian-group`. The polynomial $x + 1_{\mathbb{Z}[x]}$ is an element of the domain $\mathbb{Z}[x]$. Notice that there is a complete type hierarchy sitting above the class `polynomial-ring`.

There are always two functions for getting hold of a domain. The one that begins "`make-`" always creates a new domain. The one begining with "`get-`" tries to find an isomorphic domain that already exists. Thus `(get-rational-integers)` always returns the same rational integer domain, while `(make-rational-integers)` always creates a new instance of the rational integers.

3.2 Morphisms

Morphisms are maps between domains. They are first class objects in Weyl and can manipulated like domains and domain elements. In particular, two morphisms can be composed using the operation `compose`. Morphisms are created using the function `make-morphism` which takes three arguments.

`make-morphism` *domain mapping range*

This function creates a morphism from *domain* to *range*. *Mapping* is a function of one argument that takes an element of *domain* and returns an element of *range*.

All morphisms created are remebered in the in the `*morphisms*` data structure. (Initially, this is just a list.) To find any existing homomorphisms we can use the function `get-morphisms`.

`get-morphisms` &key *domain range*

When given no arguments this function returns a list of all the morphisms Weyl currently knows about in this context. When the *domain* is provided, it returns all morphisms from *domain* to anywhere. Similarily, when *range* is provided, only those morphisms that map into *range* are returned. When both *range* and *domain* are given, then only those maps from *range* to *domain* are returned.

Classes have been provided to indicate that more is known about the map than that is a simple morphism. In partiuclar, a morphism can be a homomorphism, an injection, a surjection, an isomorphism and an automorphism. There are

3.3 Coercions

Explicit coercions are performed by the function `coerce`:

`coerce` *element domain*

Using the canonical mapping, finds an element of *domain* that corresponds with *element*. (At this point in time, it is unlikely that we have included all of the canonical maps. Those which are implemented should be mentioned here.)

If the the switch `*allow-coercions*` is set to false (`nil`) then the canonical maps are the only ones that will be used by `coerce`. However, if the user sets `*allow-coercions*` to `T` then by creating a *homomorphism* between two domains, the set of canonical maps between domains can be extended. If `coerce` cannot find any other predefined mapping between the domain of *element* and *domain* it the searches the set of all defined homomorphisms. If there exists a *canonical homomorphism* between the two domains then it is used to map *element* into *domain*. If there does not exist a canonical homomorphism but there is exactly one homomorphism between the two domains, then it is used. If there exists a only one homomorphism between these two

As a general rule, Weyl does not provide for implicit coercions of arguments to functions. Thus we assume that in the expression `(+ a b)` the domains of `a` and `b` are the same. If this is not the case, the user must explicitly insert a coercion. The reason for this is to deal with problems such as would arrive if `a` were $1/2$ and `b` were x. The domain of `a` is $\mathbb{Q}$, while the domain of `b` is $\mathbb{Z}[x]$. The sum cannot be represented in either domain, and in fact we have the choice of embedding it in either $\mathbb{Q}[x]$ or $\mathbb{Z}(x)$. The wrong choice could be lead to very inefficient algorithms.

However, we do make one exception. There is assumed to be a canonical homomorphism of the rational integers into every domain. If only one of arguments to one of the four basic arithmetic operations is an element of $\mathbb{Z}$, then it is automatically coerced into the domain of the other argument.

3.4 Hierarchy of Domains

The root of the domain hierarchy is the class *domain*. It provides a number of utility routines for managing the set operations and axioms associated with a domain. Most all algebraic domains are built on the higher level class *Set*.

`=` x y

Test to see if x and y are equal. Equality is meant in the mathematical sense, so two vectors are `=` if their components are `=`.

The set S is a `semigroup` if there is a binary operation `times` that maps two elements of S into S.

`times` x y

Returns the product of two elements of a `semigroup`.

Products of more then two elements can be abbreviated using `*`, where

$$\texttt{(* a b c)} = \texttt{(plus (plus a b) c)}$$

In addition, repeated multiplication can be indicated by `expt`.

`expt` a n

Multiplies a by itself n times. n must be a positive Lisp integer.

`Monoid`'s are semigroup's that have an element that serves a multiplicative identity. This element is called `one` and may be accessed using the function `one`.

`one` *semigroup*

Returns the multiplicative identity of `monoid`.

`1?` *elt*

Returns true if its argument is the multiplicative identity and false otherwise.

In addition the `expt` function is extended so if its second argument is 0, it returns the multiplicative identity.

The elements of a `Groups` have multiplicative inverses which can be determined using `recip`

`recip` *elt*

Returns the multiplicative inverse of *elt*.

In a group, the `expt` operation is extended to apply to negative second arguments also. In this case

$$\texttt{(expt x } -n\texttt{)} = \texttt{(expt (recip x) } n\texttt{)}$$

A binary operation $\oplus$ on a semigroup S is *commutative* if for every pair of elements in a and b in S, $a \oplus b = b \oplus a$. The domains `abelian-semigroup`, `abelian-monoid` and `abelian-group` are similar to the domains `semigroup`, `monoid` and `group` except the binary operation is `plus` instead of `times`.

`plus` *x y*

Returns the sum of *x* and *y*, which are elements of an `abelian-semigroup`. `Plus` is a commutative binary operation.

The multiple argument version of `plus` is `+`, which is what should be used in all applications.

`zero` *abelian semigroup*

Returns the additive identity of an abelian semigroup.

`0?` *elt*

Returns true if its argument is the additive identity and false otherwise.

`minus` *elt*

Returns the additive inverse of *elt* as an element of an abelian group.

`characteristic` *ring*

Returns the characteristic of *ring*. This only applies to rings with units.

3.5 Hierarchy of Conditions

What are the conditions that are raised by the algebra package. All error conditions should be replaced by conditions. Part of the conditions should be able to answer the question "what does this error mean?" Need to work out the hierarchy of conditions. In the current system there will be very few conditions, since most of what we have is fairly low level.

Chapter 4

General Expressions

Most sophisticated computations using Weyl take place within one of the more specialized domains discussed in the later chapters. However, none of these specialized domains can be represent all mathematical objects. For instance, vectors do not fit naturally into polynomial rings. A "domain" that can represent any mathematical object is the *general expression* domain. Elements of this domain are represented as lists so it is very easy to code routines that walk through mathematical expressions (*e.g.* for simplification, integration or series expansion).

The variable `*general*` is always bound to the domain of general expressions.

4.1 Variables

Variables are have the following form

variable	:=	`(VARIABLE :SYMBOL` *variable symbol* `[:SUBSCRIPTS` *list of subscripts*`]` `[:SUPERSCRIPTS` *list of subscripts*`] [:LANGUAGE` *language*`])`
variable symbol	:=	*atom* | *symbol description* | *special char*
special char	:=	`:NABLA` |
language	:=	`:ROMAN` | `:GREEK`

Examples:

x	$\Longrightarrow$	`(VARIABLE :SYMBOL X :STRING "x")`
$x_{1,2}$	$\Longrightarrow$	`(variable :SYMBOL X :STRING "x" :SUBSCRIPTS (1 2))`
$x_1^{(n)}$	$\Longrightarrow$	`(VARIABLE :SYMBOL X :STRING "x" :SUBSCRIPTS (1)` `:SUPERSCRIPTS ((VARIABLE :SYMBOL N :STRING "n")))`

`ge-variable?` *x*

Returns `t` if x is a general expression that is a variable.

`add-subscripts` *var* &rest *subscripts*

Creates a new variable, which has the subscipts indicated. If the variable already has subscripts, then the new subscripts are appended to the ones already present.

All variables have property lists which can be used to store important information about them. This information is not attached to the variables, but to the domain from which the variables arise.

`get-variable-property` *domain var key*

Returns the *key* prorpety of *var*. A return value of `nil` indicates that the was no *key* property associated with *var*. New properties may be established by using `setf` on this form.

Dependencies of one variable on another can be declared using `declare-dependencies`:

`declare-dependencies` *kernel* &rest *vars*

This indicates that *kernel* depends upon each of the variables in `vars`.

`depends-on?` *exp* &rest *vars*

This predicate can be applied to any expression, not just to variables. It returns `T` if the `exp` depends on *all* of the variables in *vars*, otherwise it returns `NIL`.

`different-kernels` *exp list-of-kernels*

Returns a list of the kernels in *exp* that are different from those in *list-of-kernels*.

4.2 Numbers

Should say something intelligent here. In particular need to deal with arbitrary precision floating point numbers as well as the IEEE floating point standards. At the moment, we just accept floating point numbers however they are printed out.

4.3 Operators

The basic boolean operations are supported:

$a+b+c$	`(plus a b c)`
$a-b$	`(difference a b)`
$-a$	`(minus a)`
ab	`(times a b)`
a/b	`(quotient a b)`
a^b	`(expt a b)`

In addition the standard special functions are supported log sin cos tan cot sec csc sinh cosh tanh coth sech csch

4.4 Tools

4.4.1 Combinatorial Tools

Weyl provides control abstractions that are often useful for computations that involve combinatorial enumeration and searching. These control structures are `permute` and `choose` which enumerate all permuations of a sequence and all subsets of a fixed size of a sequence.

`permute` *sequence (var . options)* &body *body*

> *Sequence* is a sequence of elements (RZ: at the moment this control structure is only implemented for lists). The variable *var* is repeated bound to the different permuations of *seq* and *body* is evaluated each time. The options are provided to specify even and odd permutations, but are not at this point implemented.

As an example, the following code will print all permutations of the list `(a b c)`, and count their number:

```
> (let ((count 0))
  (permute '(a b c) (p)
    (print p)
    (incf count))
  (format t "~%~D permutations total~%" count))
(C B A)
(B C A)
(C A B)
(A C B)
(B A C)
(A B C)
6 permutations total
NIL
```

`choose` *set (var n . options)* &body *body*

> While, the first argument to `permute` must be ordered, the first argument to `choose` need only be a set (but again, only lists are currently implemented). The variable *var* is bound to each subset of *set* that has precisely n elements and *body* is evaluated in each case. At the moment no options are permitted.

4.4.2 Display tools

`display` *expr* &optional *(stream* `*standard-output*`*)*

> Prints the expression *expr* onto *stream*. If stream a graphics stream then a two dimension display will be used (not yet implemented), otherwise some textual display will be used.

4.4.3 Simplification Tools

`simplify` *expr*

> Performs simple simplifications of *expr*, $0 + x \rightarrow x$ and so on.

`ge-great` *x y*

To speed up operations like simplification of expressions, an order is placed on all expressions in the general representation. This ordering is provided by the function `ge-great`.

`def-ge-operator` *operator* &rest *keyword-expr-pairs*

When a new operator is introduced this definer should be used. It allows one to define the simplifer, display and equality tester functions.

`deriv` *exp var* &rest *vars*

Computes the derivative of *exp* with respect to *var* and simplifies the results. This is done for *var* and each element of *var*. Thus the second derivative of *exp* with respect to `t` could be computed by: `(deriv` *exp* `'t 't)`.

`declare-derivative` *func args* &body *body*

This form is used to define the derivative of *func*. The body of this form can do any computation it wants on the arguments of *func*. For instance, the derivative of sin and cosine are defined as follows:

```
(declare-derivative sin (x) var
  (* (deriv x var) (cos x)))

(declare-derivative cos (x) var
  (* (- (deriv x var)) (sin x)))
```

Chapter 5

Scalar Domains

Weyl provides four different types of primitive scalar domains: the rational integers, finite fields, the real numbers and Lisp numbers.

5.1 Rational Integers

The rational integers are the integers of elemetary arithmetic:

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}.$$

Other than limitations on the memory of the host computer, there is no limitation on the size of the elements of $\mathbb{Z}$. A domain of rational integers is created by the following function.

`get-rational-integers`

Creates a domain that is isomorphic to the rational integers.

Most of the time, there only needs to be one rational integer domain.

Rational integers are most easily created by coercing a Lisp integer to a rational integer domain using the function `coerce`. Furthermore, the usual arithmetic routines (`+`, `-`, `*`, `/` and `expt`) work with rational integers. For instance the following routine could be used to compute factorials.

```
(setq *Z* (get-rational-integers))

(defun factorial (n)
  (if (< n 2) (coerce 1 *Z*)
      (* (coerce n *Z*) (factorial (- n 1)))))
```

One of the more commonly used control structures is that used to exponentiate a number by repeated squaring. This control structure is captured by the internal function `weyli::repeated-squaring`.

`weyli::repeated-squaring` *mult one*

Returns a function of two arguments which is effectively

```
(lambda (base exp)
  (declare (integer exp))
  (expt base exp))
```

except that the body does the exponentiating by repeated square using the operation *mult*. If `exp` is 1, then *one* is returned.

Using this function, on could have defined exponentiating as

```
(defun expt (x n)
  (funcall (weyli::repeated-squaring #'times (coerce 1 (domain-of x)))
           x n))
```

However, this routine can be used for operations other than exponentiation. For instance, if one wanted a routine that replicates a sequence `n` times, one could use the following:

```
(defun replicate-sequence (x n)
  (funcall (weyli::repeated-squaring #'append ())
           x n))
```

`isqrt` n

Returns the integer part of the square root of n.

As in Common Lisp there is a set of four functions for truncating numbers and ratios to integers. If the second argument is not provided then it defaults to 1. If only argument is provided and it is a rational integer, then all four functions return the same values.

`floor` *number* &optional *divisor*
`ceiling` *number* &optional *divisor*
`truncate` *number* &optional *divisor*
`round` *number* &optional *divisor*

`weyli::power-of?` *number* &optional *base*

Returns *base* and k if $number = base^k$, otherwise it returns `nil`. If *base* is not provided returns the smallest integer of which *number* is a perfect power.

`factor` n

Factors n into irreducible factors. The value returned is a list of dotted pairs. The first component of the dotted pair is the divisor and the second is the number of times the divisor divides n. The type of factorization method used, can be controlled by setting the variable `*factor-method*`. The allowable values are `ssimple-integer-factor` and `fermat-integer-factor`.

`totient` n

Returns the Euler totient function of n, the number of positive integers less than n that are relatively prime to n, *i.e.*:

$$\texttt{totient}(n) = n \prod_p \left(1 - \frac{1}{p}\right),$$

where product is over all prime divisors of n.

(Need to point out that the elements of the second rational integer domain created are totally different from that those that are elements of the first instance of the rational integers.)

5.2 Real Numbers

`get-real-numbers` &optional *precision*

This returns a domain whose elements are floating point numbers. If *precision* is not specified, then the machines default double precision floating point numbers will be used. If *precision* is specified, then a special arbitrary precision floating point package will be used. Operations with these numbers will be somewhat slower (and will cause more garbage collection) than when using the machine's floating point data types.

`floor` *number* &optional *divisor*

Computes the floor of *number*.

`ceiling` *number* &optional *divisor*

Computes the ceiling of *number*.

`truncate` *number* &optional *divisor*

Computes the truncate of *number*.

`round` *number* &optional *divisor*

Computes the round of *number*.

`sqrt` n

For positive n returns positive $\sqrt{n}$ with the same precision as n.

The following standard trigonometric and hyperbolic routines are provided

`sin` n	`asin` n	`sinh` n	`asinh` n
`cos` n	`acos` n	`cosh` n	`acosh` n
`tan` n	`atan` n	`tanh` n	`atanh` n

`exp` n

Returns e^n.

`log` n

For positive n returns the principal part of $\log_e n$.

In addition following two versions of the logarithm function are also provided.

`log` n

For positive n returns the principal part of $\log_e n$.

`log2` n b

For positive n returns the principal part of $\log_b n$.

5.3 Complex Numbers

Not yet implemented.

`realpart` *z*

If $z = x + iy$ returns x.

`imagpart` *z*

If $z = x + iy$ returns y.

`conjugate` *z*

If $z = x + iy$ returns $x - iy$.

`abs` *z*

If $z = x + iy$ returns $|z| = \sqrt{x^2 + y^2}$.

`phase` *z*

If $z = re^{i\theta}$ returns θ.

5.4 Finite Fields

The usual finite fields are provided in Weyl, $\mathbb{F}_p$ and algebraic extensions of $\mathbb{F}_q$. Such domains are called *GFp* domains. Since all finite fields with the same number of elements are isomorphic, fields are created by specifying the elements in the field.

`get-finite-field` *size*

Size is expected to be a a power of a prime number. This function returns a finite field with the indicated number of elements.

`number-of-elements` *finite-field*

Returns the number of elements in *finite-field.*

At the moment Weyl can only deal with the fields $\mathbb{F}_{2^k}$ and F_p. For instance,

```
> (setq F256 (get-finite-field 256))
GF(2^8)

> (characteristic F256)
2

> (number-of-elements F256)
256
```

Elements of a GFp are created by coercing a rational integer into a GFp domain. For finite fields with characteristic greater than 2, coercing an integer into F_p maps n into $n \pmod p$. For F_{2^k}, the image of an integer is a bit more complicated. Let the binary representation of n be

$$n = n_\ell n_{\ell-1} \cdots n_2 n_1 n_{0\,2},$$

and let α be the primitive element of F_{2^k} over F_2. Then

$$n \longrightarrow n_{k-1}\alpha^{k-1} + n_{n-2}\alpha^{k-2} + \cdots + n_1\alpha + n_0.$$

This mapping is particularly appropriate for problems in coding theory.

In addition, elements of finite fields can be created using `make-element`.

`make-element` *finite-field integer* &optional *rest*

Creates an element of *finite-field* from *integer*. This is the only way to create elements of $\mathbb{F}_{p^k}$
$\mathbb{F}_p$.

As an example of the use of finite fields, consider the following function, which determines the order of an element of a finite field (the hard way).

```
(defun element-order (n)
  (let* ((domain (domain-of n))
         (one (coerce 1 domain)))
    (loop for i upfrom 1 below (number-of-elements domain)
          for power = n then (* n power)
        do (when (= power one)
             (return i)))))
```

A more efficient routine is provided by Weyl as `multiplicative-order`.

`multiplicative-order` *elt*

Elt must be an element of a finite field. This routine computes multiplicative order of *elt*. This routine requires factoring the size of the multiplicative group of the finite field and thus is appropriate for very large finite fields.

The following illustrates use of these routines.

```
> (element-order (coerce 5 (get-finite-field 41)))
20

> (multiplicative-order (coerce 5 (get-finite-field 41)))
```

Consider what is involved when implementing an algorithm using the Chinese remainder theorem. The computation is done in a number of domains like $\mathbb{Z}/(p_1)$, $\mathbb{Z}/(p_2)$ and $\mathbb{Z}/(p_3)$. The results are then combined to produce results in the domains $\mathbb{Z}/(p_1p_2)$ and $\mathbb{Z}/(p_1p_2p_3)$. Rather than working in several different domains and explicitly coercing the elements from one to another, it is easier to assume we are working in a single domain that is the union of $\mathbb{Z}/(m)$ for all integers m and marking the elements of this domain with their moduli. We call this domain a *GFm*.

GFm domains are also created using the `get-finite-field` but by providing `nil` as the number of elements in the field.

We denote a element of a GFm by indicating their modulus in a subscript with parentheses. Thus $2_{(5)}$ means 2 modulo 5. Combining two elements $a_{(m)}$ and $b_{(m)}$ that have the same moduli is the same as if they were both elements of $\mathbb{Z}/(m)$. To combine elements of two different rings, we find a ring that contains both as subrings and perform the calculationt there. Thus combining $a_{(m)}$ and $b_{(n)}$ we combine the images of a and b as elements of $\mathbb{Z}/(\gcd(m, n))$.

FIXTHIS: Need works something out for dealing with completions of the integers at primes, and how we are going to compute with elements.

5.5 Lisp Numbers

The `lisp-number` domain is a special domain that is used for efficiency. Its elements are represented as Lisp integers, ratios, floating point numbers or complex floating point numbers. The native Lisp arithmetic operations are used on them for efficiency. This scalar domain should not be needed by most users.

`weyli::get-lisp-numbers`

Creates a domain that is isomorphic to the rational integers.

All of the usual Lisp routines for dealing with Lisp numbers are available in Weyl.

One change between Common Lisp and Weyl is that we have modified the printer, so that Lisp complex numbers are printed as `(`a` + `b` i)` or just a, if the imaginary part is zero.

Chapter 6

Polynomial Rings

6.1 Creating Polynomial Rings

Polynomial rings are domains which consist of polynomials in some number of variables over a coefficient domain, e.g. $\mathbb{Z}[x, y]$, $\mathbb{R}[x]$, $\mathbb{Q}((t))[x]$. In these three cases the coefficient domains are the rational integers $\mathbb{Z}$, the real numbers $\mathbb{R}$ and powerseries' in t. Polynomial rings are created using the function `get-polynomial-ring`.

`get-polynomial-ring` *coefficient-domain list-of-variables*

> *Coefficient-domain* is must be a ring. *List-of-variables* is a list of arbitrary Lisp objects, each of which represents a generator of the polynomial ring.

`transcendence-degree` *domain1 domain2*

> This function checks to see that *domain2* is a subring *domain1*, and if so returns the transcendence degree of *domain1* over *domain2*.

6.2 Information About Variables

The variables of a polynomial ring have three different representations: as a polynomial, as symbol, or as a *variable index*. The simplest representation is as a symbol. These are the symbols passed to `get-polynomial-ring` and are the only mechanism the user has to influence the external printed representation of the variable. Internal to the polynomial package, there are (Lisp) integers associated with variables. These integers are used as indices into tables and to indicate the ordering of variables in multivariate polynomials. Finally, a variable can be represented as a polynomial.

Regardless of the internal representation used for polynomials, there is an integer associated with each variable of the polynomial. These numbers are used as an index into all data structures that hold additional information about the variable. The index corresponding to a variable is obtained by the generic function `variable-index`. The variable correspoding to an order number is `variable-symbol`.

`variable-index` *domain variable*

Returns the variable index corresponding to *variable*. *Variable* can be either a symbol or a polynomial.

`variable-symbol` *domain variable*

Returns the variable symbol corresponding to *variable*. *Variable* can be either an integer or a polynomial.

There is a property list associated with each variable. This property list is accesed using the generic function `get-variable-property`. Properties can modified using `setf`.

`get-variable-property` *domain variable property*

Returns a *property* property of `variable`.

`add-new-variable` *ring var*

Var is a variable in the general representation. This function modifies *ring* to have an addition variable.

6.3 Polynomial Operators

`scalar?` *polynomial*

The argument of this function is expected to be an element of a polynomial domain. If the argument is an element of the coefficient field this function returns `T` otherwise it returns `NIL`

`degree` *polynomial var*

Returns the degree of *polynomial* in the variable *var* as a lisp integer.

The usual arithmeic operations including `plus`, `minus`, `difference`, `times`, `quotient`, `recip` and `expt`, can be applied to polynomials. For polynomial rings, `expt` is restricted to integer exponents.

`remainder` *x y*

Computes the polynomial pseudo-remainder of `x` and `y`.

`gcd` *x y*

Computes the polynomial greatest common divisor of `x` and `y`. The variable `weyli::poly-gcd-algorithm` is bound to the particular gcd algorithm to be used.

`partial-deriv` *polynomial variable*

This function takes the partial derivative of *polynomial* with respect to *variable*. *Variable* is actually a polynomial, not the lisp object which is the variable

`deriv` *polynomial var* &rest *vars*

Computes the derivative of *exp* with respect to *var*. This is done for *var* and each element of *var*. The variables can either be elements of `*general*` or of a polynomial domain.

`coefficient` *poly var* &optional *degree*

Compute the coefficient of var^{degree} in *poly*. *degree* defaults to 1.

`substitute` *value var poly*

Substitutes *value* for each occurence of *var* in *poly*. If *value* is a list, it is interpreted as a set of values to be substituted in parallel for the variables in *var*. The values being substituted must be either elements of the domain of *poly* or its coefficient domain.

`list-of-variables` *poly* &optional *vars*

Returns a list of the variables that actually appear in *poly*. If provided *vars* must be a list. The variables that appear in *poly* are added to *vars*.

6.4 Gröbner Bases

(RZ: Everything in this section is preliminary. I have not worked out a satisfactory interface to the Gröbner basis code.)

A set of polynomials can be collected into an object called a `grobner-basis`.

`make-ideal` &rest *polys*

Creates the ideal generated by the polynomials *polys*. The polynomials must all come from the same ring. Initially, we have only implmented the Gröbner basis algorithm for polynomials over fields, the domain of the polynomials must be a polynomial ring over a field.

6.5 Differential Rings

Differential rings are polynomial rings that have *derivations*. A derivation of a differential ring $R\langle x\rangle$ is a map δ from $R\langle x\rangle$ to $R\langle x\rangle$ which has the following properties:

$$\begin{aligned} \delta(ap + bq) &= a\delta p + b\delta q \\ \delta a &= 0 \\ \delta(pq) &= p\delta q + q\delta p \end{aligned}$$

where $a, b \in R$ and p and q are not elements of R.

`get-differential-ring` *domain variables*

Return a differential ring whose coefficients lie in `domain` and which contains the variables `variables`. For example:

```
=> (setq R (get-differential-ring (get-rational-integers) '(x y)))
Z<x, y>
```

`derivation` *p*

Returns the derivative of the differential polynomial p. For exmaple:

```
=> (setq p (expt (+ (coerce 'x r) (coerce 'y r)) 2))
x^2 + 2 y x + y^2

=> (deriv p)
(2 D{x, 1} + 2 D{y, 1}) x + ((2 D{x, 1} + 2 D{y, 1}) y)
```

`variable-derivation` *domain variable*

Returns the derivative of *variable*. The derivative can be either an element of `domain`, or `:generate` which indicates that the derivative will be a new variable which is yet to be created.

The derivative of a variable can be set using `setf`.

6.6 Structure Types for Polynomials

Polynomials are currently represented using one of three different structure types. The simplest structure is only used by elements of univariate polynomial rings and is called `weyli::upolynomials`. Two different representations are provided for multivariate polynomials. The `weyli::mpolynomial` structure type uses a recursive structure, so polynomials can be views as univariate polynomials with polynomial coefficients. This is the classical representation used by systems like Macsyma. The `weyli::epolynomial` structure is represents polynomials as a set of pairs of exponent vectors and coefficients.

6.6.1 Univariate Polynomials

6.6.2 Multivariate Polynomials

6.6.3 Expanded Polynomials

`weyli::make-epolynomial` *domain poly-form*

Generates a polynomial in *domain* with structure type `weyli::epolynomial` that corresponds to *poly-form*.

6.7 The Spectrum of a Ring

`Spectrum` *Ring*

This function returns a new domain which is the spectrum of *ring*. The elements of this domain are the ideals of *ring*. (Need to figure out how the topology of Spec R fits into all this.)

Chapter 7

Algebraic Extensions

`degree` *domain1 domain2*

This function checks to make sure that *domain2* is a subring of *domain1*, and if so it returns the degree of `domain2` over `domain1`

7.1 Algebraic Closures

Need to special case the algebraic closure of the real numbers. This is the only algebraic closure that is of finite degree.

Chapter 8

Sums, Products and Quotients of Domains

Let $\mathfrak{A}$ be category of domains, *e.g.*, groups, rings, fields, etc. Given two elements of $\mathfrak{A}$, say A and B, there are often ways of combining them to produce another element of $\mathfrak{A}$. This chapter discusses several of those techniques.

8.1 Direct Sums

Given two domains A and B, their direct sum can be viewed as the set of all pairs of elements in A and B, where arithmetic operations are performed componentwise. We denote the direct sum of A and B by $A \oplus B$. If A, B and C are domains, then

$$A \oplus (B \oplus C) = (A \oplus B) \oplus C = A \oplus B \oplus C,$$

so multiple direct sums are well defined. Direct sums are created using the function `get-direct-sum`:

`get-direct-sum` domain$_1$ ... domain$_n$

> Creates a domain that is the direct sum of $domain_1, \ldots, domain_n$. If any of the $domain_i$ are themselves direct sums, then their components are included as if they had been explicitly specified.

The algebraic structure of the object returned by `get-direct-sum` depends upon the type of the objects of which it is composed. The following table illustrates the options.

A	B	$A \oplus B$
Group	Group	Group
Ring	Ring	Ring
Field	Field	Ring
Group	Ring	Group
Abelian	Abelian	Abelian

The dimension of a direct sum is the number of its components:

`dimension` *direct-sum*

Returns the number of components of the direct sum.

Elements of direct sums are created using `make-element`

`make-element` *direct-sum* $\text{element}_1 \ldots \text{element}_n$

Creates an element of *direct-sum*. The number of elements provided must match the dimension of *direct-sum*.

The following illustrates the use of direct sums. First we create a direct sum domain. The symbol used for a direct sum when only ascii characters are available is `(+)`.

```
> (setq sum (get-direct-sum (get-rational-integers) (get-finite-field 5)))
Z (+) GF(5)
```

Next we create an element of `sum`. Notice that we can use the canonical coerces of Lisp integers into the finite field.

```
> (setq x (make-element sum 2 3))
2 (+) 3(5)

> (+ x x)
4 + 1(5)
```

The dimension of the direct sum domain and its components are easily determined using `dimension` and `ref`.

```
> (dimension sum)
2

> (loop for i below (dimension sum)
        do (format t "~%Component ~D: ~S" i (ref sum i)))
Component 0: Z
Component 1: GF(5)
```

Notice that `ref` also works with the elements of `sum`

```
> (ref x 1)
3(5)
```

8.2 Tensor Products

8.3 Quotient Fields

`make-quotient-field` *domain*

This generic function constructs the quotient field of *domain*. The domain returned by this operation will be a field. If *domain* is itself a field, then it will be returned without any modification. If *domain* is a gcd domain then operations with the elements of the resulting quotient field will reduce their answers to lowest terms by dividing out the common gcd of the resulting numerator and divisor.

Two special cases are handled specially by `make-quotient`. If the argument domain is either the rational integers or a polynomial ring then special domains of the rational numbers or rational functions are used. This is for efficiency reasons. Hopefully, a more general solution can be found in the near future.

Two operations can be used to create elements of a quotient field: `make-quotient-element` and `quotient-reduce`. The operation `make-quotient-element` creates a quotient element from the numerator and denominator domains. `quotient-reduce` does the same thing but removes the common GCD from the numerator and denominator first.

(Ed: Need to think about things like localizations here.)

`get-quotient-field` *ring*

Returns a field which is the quotient field of *ring*. In some cases, this is special cased to return return something more efficient than the general quotient field objects.

`weyli::qf-ring` *qf*

Returns the ring from which the quotient field *qf* was built.

`numerator` *q*

Returns the numerator of *q*.

`denominator` *q*

Returns the denominator of *q*.

`quotient-reduce` *domain numerator* &optional *denominator*

Numerator and *denominator* are assumed to be elements of base ring of *domain*. `quotient-reduce` creates and quotient element in *domain* from *numerator* and *denominator*. If *denominator* is not provided, the multiplicative unit of *domain* is used.

`with-numerator-and-denominator` *(num den) q* &body *body*

Creates a new lexical environment where the variables *num* and *den* are bound to the numerator and denominator of *q*. Each

Chapter 9

Spaces

Given a field k we can construct multidimensional *spaces* whose elements are tuples, where each component is an element of k. Currently we construct two different types of spaces, *affine* spaces and *projective* spaces. Elements of an affine space of dimension n, which are called *affine points* have n components. Two affine points are equal if their corresponding components are equal. Elements of a projective of dimension of n have $n+1$ components. Two projective points u and v are equal if there is a non-zero element of k, α such that $\alpha u = v$.

Both affine and projective spaces are algebraically modules. Their coefficient domain is the base ring.

`dimensions` *space*

> Returns a list of the dimensions of the space. For free-modules, vector spaces, projective spaces, and so on there is only one element in this list. However for spaces like $\mathbb{R}^3 \times \mathbb{Z}^5$ there may be more elements.

(Ed: At the moment affine spaces are called vector spaces, and their elements are called vectors. I recommend that we change this to calling them affine spaces and their elements points, as I've documented here.)

(Ed: What do we do about the topology of the space?)

9.1 Point Set Topology

Objects in topologies are subsets of *spaces*. The root class of all spaces is `abstract-space`. Abstract spaces do not have a dimension and consist only of points. The class `space` is an abstract space that has a dimension.

(ed: Need to put in all the topological information, Hausdorf, compact, connected, etc.)

Points in a space are created using `make-element`:

`make-element` *space coord1* &rest *coords*

> For an abstract space the values of the coordinates are ignored and a new point is created.

All points in a space are ordered. This is currently implemented by have an slot in each point that contains an integer. This integer is unique for each point. For some spaces, *e.g.* $\mathbb{R}$, it may be more appropriate to use a different ordering.

```
> (setq ab-space (make-instance 'abstract-space))
#<ABSTRACT-SPACE>

> (progn
    (setq a (make-element abs-space nil)
          b (make-element abs-space nil)
          c (make-element abs-space nil))
    (list a b c))
(<1> <2> <3>)
```

9.2 Algebraic Topology

9.2.1 Simplices

Simplices are collections of points in independent points in a space. If the space has dimension n, then a simplex can contain no more than $n + 1$ points. Simplices are created with `make-simplex`

`make-simplex` $\mathrm{vertex}_0 \ldots vertex_k$

Creates a k simplex whose components are $\mathrm{vertex}_0, \ldots, \mathrm{vertex}_k$. This routine ensures that all of the points are from the same space, and if the space has coordinates, that the points are independent. The points in the simplex are ordered according to the point ordering of the space. Simplices are immutable, *i.e.*, once created they can not be modified.

Thus a triangle would be created:

```
> (setq triangle (make-simplex a b c))
[<A>, <B>, <C>]
```

Two simplices are equal if they have the same vertex set.

```
> (= triangle (make-simplex a b c))
t
```

`vertex-set` *simplex*

Returns a list of the vertices of *simplex*. For instance,

```
> (vertex-set triangle)
(<A> <B> <C>)
```

Notice that there aren't any commas between the vertices. This is a real list not a simplex.

`subset?` *simplex1 simplex2*

Returns `t` if *simplex1* is a subset of *simplex2*. That is, if each of the vertices of *simplex1* are elements of the vertex set of *simplex2*.

`map-over-simplices` *simplex (var n)* &body *body*

var is bound to each sub-simplex of *simplex* of dimension n, and the forms in *body* are then evaluated. If n `nil` then all sub-simplices of any dimension are used.

```
> (map-over-simplices triangle (simp nil)
    (print simp))
[<A>]
[<A>, <B>]
[<A>, <C>]
[<A>, <B>, <C>]
[<B>]
[<B>, <C>]
[<C>]
```

9.2.2 Complexes

A complex $\mathcal{C}$ is a collection of simplices of the same space with the property that if A is in $\mathcal{C}$ then any subset of A will also be an element of $\mathcal{C}$.

`make-complex` $simplex_1 \ldots simplex_n$

Creates a simplex that consists of each of the $simplex_i$'s and each of the sub-simplices.

```
> (setq complex (make-complex triangle))
#<COMPLEX>
```

`vertex-set` *complex*

Returns a list of the elements of each of the 0-simplices in *complex*.

Unlike simplices, complexes are mutable. New simplices can be added and deleted from them using `insert` and `delete`.

`insert` *simplex complex* &key *(subsimplices?* `t`*)*

Adds *simplex* to complex if it is not already an element. If *subsimplicies?* is true, which it is by default, then all of the subsimplices of *simplex* will also be added. If *subsimplices?* is false, then the resulting object not be a true complex. This option is provided for internal routines.

`delete` *simplex complex* &key *(subsimplices?* `t`*)*

Deletes *simplex* from complex if it is an element. If *subsimplicies?* is true, which it is by default, then all of the subsimplices of *simplex* will also be deleted. If there is a simplex of *complex* that contains *simplex* then the resulting object will not be a complex!

If *subsimplices?* is false, then the resulting object not be a true complex. This option is provided for internal routines.

`map-over-simplices` *complex (var n)* &body *body*

var is bound to each sub-simplex of *simplex* of dimension n, and the forms in *body* are then evaluated. If n `nil` then all sub-simplices of any dimension are used.

9.2.3 Homology

If $\mathcal{K}$ is a complex then an p-chain over an abelian group G is a map from each of the p-simplices in K to an element of K that vanishes on all but finitely many p-simplices and which takes on opposite values for simplices with different orientations. That is if c_p is a p-chain and σ and σ' are p-simplices of $\mathcal{K}$ with the same vertex sets, but with different orientations, then

$$c_p(\sigma') = -c_p(\sigma).$$

Denote by c_σ the p-chain which has the properties:

$$c_\sigma(\tau) = \begin{cases} 1 & \text{if } \tau = \sigma \\ -1 & \text{if } \tau \text{ has same vertices as } \sigma\text{, but different orientation} \\ 0 & \text{otherwise} \end{cases}$$

By abuse of notation, we write σ in place of c_σ so that $\sigma(\sigma) = 1$. Then every p chain c can be written as a finite sum of the form:

$$c = c_1\sigma_1 + \cdots + c_\ell\sigma_\ell.$$

where $c_i \in G$ and σ_i are p-simplices in $\mathcal{K}$. The set of p chains forms a module under addition and is denoted by $C_p(\mathcal{K}; G)$ or $C_p(K)$ if $G = \mathbb{Z}$.

`get-chain-module` *complex n* &optional *group*

Creates the group of n chains of *complex* with coefficients in *group*. If *group* is not provided then $\mathbb{Z}$ is used instead. If provided, *group* must be an abelian group.

Individual chains may be created by coercing a simplex into the chain module.

```
> (setq 1-chains (get-chain-module 1 complex))
C[1](#<COMPLEX>)

> (coerce (make-simplex a b) 1-chains)
[<A>, <B>]

> (+ (coerce (make-simplex a b) 1-chains)
     (* 2 (coerce (make-simplex b c) 1-chains)))
[<A>, <B>] + 2[<B>, <C>]
```

If $\sigma = [v_0, \ldots, v_p]$ is a p-simplex, then its boundary is the following $p-1$ chain:

$$\partial\sigma = \partial[v_0, \ldots, v_p] = \sum_{i=0}^{p}(-1)^i[v_0, \ldots, \hat{v}_i, \ldots, v_p],$$

where $\hat{v}_i$ indicates that the v_i vertex is missing. The boundary operation can be extended to chains by linearity.

`boundary` *chain*

chain can be either chain or a simplex. It returns the chain representing the boundary of *chain*.

This is illustrated by computing the boundary of `triangle`.

```
> (setq tri-bound (boundary triangle))
[<A>, <B>] + [<B>, <C>] - [<A>, <C>]

> (boundary tri-boundary)
0
```

It is generally true that $\partial^2\sigma = 0$ for all simplices σ.

9.3 Free Modules

A module M over a ring A is an abelian group (under `+`) with an operation (`*`) such that for $a, b \in A$ and $x, y \in M$, `(* a x)` $\in M$,

$$\texttt{(* (+ } a\ b\texttt{) } x\texttt{)} = \texttt{(+ (* } a\ x\texttt{) (* } b\ x\texttt{))}, \text{and}$$

$$\texttt{(* } a\ \texttt{(+ } x\ y\texttt{))} = \texttt{(+ (* } a\ x\texttt{) (* } a\ y\texttt{))}.$$

Vector spaces and rings of matrices are common examples.

`coefficient-domain` *domain*

Returns the coefficients of the module *domain*.

`get-free-module` *domain dim*

Creates a free module of dimension *dim* where the elements' components are all elements of *domain*. *Domain* must be a ring. (Ed: If *domain* is a field, should this return a vector space instead?)

`get-vector-space` *domain dim*

Create a vector space of dimension *dim* where the vectors' components are all elements of *domain*. *Domain* must be a field.

`dimension` *domain*

Returns the dimention of *domain*.

`make-element` *domain value* &rest *values*

Make an element of the module *domain*, where the first component is *value*, etc.

`ref` *vector i*

Returns the i^{th} element (zero based) of *vector*.

`dot-product` *u v*

Computes the inner (dot product) of the two vectors v_1 and v_2. If $u = (u_1, \ldots, u_k)$ and $v = (v_1, \ldots, v_k)$ then

$$\texttt{(dot-product } u\ v\texttt{)} = u_1 v_1 + u_2 v_2 + \cdots u_k v_k.$$

9.4 Affine Spaces

Affine spaces are created using the function `make-affine-space`:

`make-affine-space` *field dimension*

> Create a projective space of dimensions *dimension* where the components are elements of the field *field*. (Ed: I think this should be weaked to work for arbitrary rings.)

Elements of an spaces, both affine and projective, are created using the generic function `make-point`:

`make-element` *space* &rest *elements*

> Creates an element of *space*. *Elements* is a list of n elements which can be coerced into the coefficient domain of space.

There is also an internal function `weyli::make-element` that which does not do any checking of its arguments and can lead to rather subtle problems if used incorrectly. On the other hand it is noticably faster.

`cross-product` *u v*

> This is currently only defined for elements of three dimensional vector spaces. If $u = (u_1, u_1, u_3)$ and $v = (v_1, v_2, v_3)$ then
>
> $$(\texttt{cross-product}\ u\ \ v) = (u_2v_3 - u_3v_1, u_3v_0 - u_0v_3, u_1v_2 - u_2v_1).$$

9.5 Projective Spaces

`make-projective` *field dimension*

> Create a projective space of dimensions *dimension* where the components are elements of the field *field*.

As in the case of regular affine spaces, elements of a projective space are created using the generic function `make-point`.

`make-point` *space elements*

> Creates a point which an element of *space*. For projective spaces, *elements* can be a list of either n or $n+1$ elements of the coefficient domain of space. If $n+1$ elements are are provided then these are the full set of elements of the point. If only n elements are provided, the missing element is filled out by a 1.

Affine spaces can be embedded in projective spaces. For projective spaces of dimension n, there are $n+1$ canonical embeddings. The function `make-affine-projection` is passed a projective space and creates an affine space with an attached homomorphism into the projective space.

`make-affine-projection` *space* &optional *dimension*

This function returns an affine space that is the projection of *space* where we hold the component *dimension* fixed. For instance, let $p = (u, v, w)$ is an element of a two dimension projective space P^2. The image of p in the affine projection of P^2 with dimension 0 held fixed is $(v/u, w/u)$. With dimension 1 is held fixed then p maps to $(u/v, w/v)$. Finally, if dimension 2 is held fixed, p maps to $(u/w, v/w)$. If *dimension* is not provided then we produce an affine space where the last dimension is held fixed.

9.6 Charts

Chapter 10

Matrices

Some stuff on matrices

10.1 General Matrices

`get-matrix-space` *ring*

Returns a general matrix domain, where the entries in the matrices are elements of *ring*.

`make-element` *domain value* &rest *values*

Make an element of the module *domain*, where the first component is *value*, etc.

The following two functions return square matrices whose off diagonal elements are all zero. Its diagonal elements are 0 for `zero-matrix` and 1 for `one-matrix`.

`zero-matrix` *matrix-domain* &optional *rank*
`one-matrix` *matrix-domain* &optional *rank*

`matrix-dimensions` *matrix*

Returns the dimensions of this particular matrix.

`ref` *vector i j*

Returns the (i, j) element (zero based) of *vector*. If i or j is `:*`, then a row or column vector is returned.

`with-matrix-dimensions` *(dim1 dim2* &optional *array) matrix* &body *body*

This control abstraction binds *dim1* and *dim2* to the dimensions of *matrix*. If *array* is provided, it is bound to an array that whose elements are the elements of the *matrix*. This control structure can lead to substantially more efficient manipulation of dense matrices.

Ultimately, *array* will turn into an efficient accessor for the elements of *matrix* so that this technique will work with other than dense matrices.

`transpose` *matrix*

Transpose *matrix*.

`jacobian` *function-list var-list*

Returns the Jacobian of *function-list* with respect to *vector-list*. These should really be vectors, but list will work also.

10.2 Matrix Groups

Chapter 11

Defsystem

This section is an adaptation of the documentation provided by Doug Rand of Prime, Inc. who originally developed this `defsystem`.

11.1 Introduction

Common LISP lacks a method for tying a group of files together under a convenient name. The common method for doing this in ZetaLISP is defsystem. This is a public domain implementation of defsystem. Where possible the same keywords are used as in ZetaLISP but this implementation has a slightly different flavor from the ZetaLISP defsystem.

Changes (2.2):

`Compile-load-system` combines `compile-system` and `load-system` as a convenience.

The package name is now `defsys` to eliminate name conflicts with the defsystem macro.

All the system oriented functions (`load-system`, `compile-system` and `show-system`, `compile-load-system`) now have the system-name argument as an optional argument. If the argument is given then the variable `*current-system*` is set to that value. If the argument is ommited then the value of *current-system* is used. If `*current-system*` is `NIL` then an error is raised.

A variable `defsys::*downcase-path-from-module-name*` is set to `T` under UNIX and `NIL` otherwise. Since module names are normally entered as symbols the mapping is normally into uppercase. Since UNIX is case sensitive this means that one would then need to name the files in uppercase.

Defsystem macro:

```
(defsystem system-name
   (system-options*)
   module-descriptions*
   )
```

Load order is implied by the order of the modules. System options are (defaults in s):

:host (`nil`) If supplied, should be a string, ignored under Lucid.

:default-pathname (`#P""`) The default place to find files. (You're safer using strings than pathnames)

:default-package (`nil`) The default package to load/compile modules in.

:needed-systems (`nil`) A list of subsystems, which must be loaded when this system is loaded.

:load-before-compile (`nil`) A list of subsystems needed for compilation

A module is a single name representing a file. A module description is either a module name or a list whose car is the module name and the cdr is a set of keywords and values. The module options are:

:recompile-on This will cause the module list to be checked for dtm if one of the listed modules is newer then the current module will be recompiled. If the current module is recompiled the list of recompile dependencies will be loaded first.

This is also a recursive recompilation. If foo dependends on bar and bar is out of date then bar will be recompiled.

:load-before-compile These are modules that are loaded before recompiling the current module.

:load-after This is really a useful option only for modules during compilation since the load order will normally be satisfied during a `load-system`. These are followed until a loaded module is found.

:pathname *path* If specified it gives a pathname to find this module. Normally this defaults to the concatenation of the default pathname for the system and the module name.

:package *package-name* What package to load/compile this module in. Defaults to the system default package.

:compile-satisfies-load If `T` then compiling this module will set it's loading information to `T`. This is usually true for files with just macros. When this option is nto supplied it defaults to `nil`.

`undefsystem` *system-name*

Removes the system description from `*all-systems*`.

`load-system` *system* &key *(reload* `nil`*)* *(included-components* `t`*)*

Loads modules of a *system*. `Load-system` is called recursively for all required systems. If *reload* is `T` force a full reload of everything. If *included-components* is `T` call `load-system` on subcomponents

`compile-system` *system-name* &key *(recompile* `nil`*)* *(included-components* `t`*)* *(reload* `nil`*)*

Compiles all modules requiring recompilation. If *recompile* is `T` all recompilations to occur regardless of "need." Need is determined by the date-time of the respective binary and source files. If *recompile* is `T` then everything is recompiled. If *included-components* if `T` then `compile-system` is applied to each of the subsystems of *system*. If *reload* is `T` force a full reload of everything.

`show-system` *system*

Print a readable description of *system*'s organization.

11.2 Simple Sample Defsystems

```
(defsystem life
  (:needed-systems (curses)
   :default-pathname #P"doug.x>lisp>life>")
  life)

(defsystem curses
  (:default-pathname #P"doug.x>lisp>curses>"
   :default-package curses)
  curses
  (curses-internals :package curses-internals))

(defsystem profile
  ()
  profile timer)
```

Bibliography

[1] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Kenneth Kahn, Sonya E. Keene, Gregor Kiczales, Larry Masinter, David A. Moon, Mark Stefik, and Daniel L. Weinreb. Common Lisp object system specification. Technical Report 88–002, X3J13, ANSI Common Lisp Standardization Committee, July 1988.

[2] J. Dongarra, J. R. Bunch, Cleve B. Moler, and G. W. Stewart. *LINPACK User's Guide*. SIAM Publications, Philadelphia, PA, 1978.

[3] Anthony C. Hearn. Reduce 3 user's manual. Technical report, The RAND Corp., Santa Monica, CA, 1986.

[4] Richard D. Jenks and Barry Marshal Trager. 11 keys to new Scratchpad. pages 123–147.

[5] Maple Group, Waterloo, Canada. *Maple*, 1987.

[6] Cleve B. Moler. Matlab user's guide. Tech. Report CS81–1, Dept. of Computer Science, University of New Mexico, Albuquerque, NM, 1980.

[7] B. T. Smith, James M. Boyle, Y. Ikebe, Virginia C. Klema, and Cleve B. Moler. *Matrix Eigensystem Routines: EISPACK Guide*. Springer-Verlag, New York, NY, second edition, 1976.

[8] Guy Lewis Steele Jr. *Common Lisp, The Language*. Digital Press, Burlington, MA, 1984.

[9] Symbolics, Inc., Burlington, MA. *MACSYMA Reference Manual*, 14th edition, 1989.

[10] Steven Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, Redwood City, CA, 1988.

Index