Chapter 9

Some Examples of Domains and Packages

In this chapter we show examples of many of the most commonly used AXIOM domains and packages. The sections are organized by constructor names.

9.1 AssociationList

The `AssociationList` constructor provides a general structure for associative storage. This type provides association lists in which data objects can be saved according to keys of any type. For a given association list, specific types must be chosen for the keys and entries. You can think of the representation of an association list as a list of records with key and entry fields.

Association lists are a form of table and so most of the operations available for `Table` are also available for `AssociationList`. They can also be viewed as lists and can be manipulated accordingly.

This is a `Record` type with age and gender fields.

```
Data := Record(monthsOld :  Integer, gender :  String)
```

$$\mathrm{Record}(\mathrm{monthsOld}\colon \mathrm{Integer},\mathrm{gender}\colon \mathrm{String})$$

`Type:  Domain`

In this expression, `al` is declared to be an association list whose keys are strings and whose entries are the above records.

```
al :  AssociationList(String,Data)
```

`Type:  Void`

The **table** operation is used to create an empty association list.

```
al := table()
```

$$table()$$

```
Type: AssociationList(String,Record(monthsOld: Integer,gender: String))
```

You can use assignment syntax to add things to the association list.

```
al."bob" := [407,"male"]$Data
```

$$[monthsOld = 407, gender = \texttt{"male"}]$$

```
Type: Record(monthsOld: Integer,gender: String)
```

```
al."judith" := [366,"female"]$Data
```

$$[monthsOld = 366, gender = \texttt{"female"}]$$

```
Type: Record(monthsOld: Integer,gender: String)
```

```
al."katie" := [24,"female"]$Data
```

$$[monthsOld = 24, gender = \texttt{"female"}]$$

```
Type: Record(monthsOld: Integer,gender: String)
```

Perhaps we should have included a species field.

```
al."smokie" := [200,"female"]$Data
```

$$[monthsOld = 200, gender = \texttt{"female"}]$$

```
Type: Record(monthsOld: Integer,gender: String)
```

Now look at what is in the association list. Note that the last-added (key, entry) pair is at the beginning of the list.

```
al
```

$$\begin{aligned} table\,(&\texttt{"smokie"} = [monthsOld = 200, gender = \texttt{"female"}],\\ &\texttt{"katie"} = [monthsOld = 24, gender = \texttt{"female"}],\\ &\texttt{"judith"} = [monthsOld = 366, gender = \texttt{"female"}],\\ &\texttt{"bob"} = [monthsOld = 407, gender = \texttt{"male"}]) \end{aligned}$$

Type: AssociationList(String,Record(monthsOld: Integer,gender: String))

You can reset the entry for an existing key.

```
al."katie" := [23,"female"]$Data
```

$$[monthsOld = 23, gender = \texttt{"female"}]$$

Type: Record(monthsOld: Integer,gender: String)

Use **delete!** to destructively remove an element of the association list. Use **delete** to return a copy of the association list with the element deleted. The second argument is the index of the element to delete.

```
delete!(al,1)
```

$$table(\texttt{"katie"} = [monthsOld = 23, gender = \texttt{"female"}],$$

$$\texttt{"judith"} = [monthsOld = 366, gender = \texttt{"female"}],$$

$$\texttt{"bob"} = [monthsOld = 407, gender = \texttt{"male"}])$$

Type: AssociationList(String,Record(monthsOld: Integer,gender: String))

For more information about tables, see 9.64 on page 215. For more information about lists, see 9.36 on page 129. Issue the system command

```
)show AssociationList
```

to display the full list of operations defined by `AssociationList`.

9.2 BalancedBinaryTree

`BalancedBinaryTrees(S)` is the domain of balanced binary trees with elements of type `S` at the nodes. A binary tree is either `empty` or else consists of a `node` having a `value` and two branches, each branch a binary tree. A balanced binary tree is one that is balanced with respect its leaves. One with 2^k leaves is perfectly "balanced": the tree has minimum depth, and the `left` and `right` branch of every interior node is identical in shape.

Balanced binary trees are useful in algebraic computation for so-called "divide-and-conquer" algorithms. Conceptually, the data for a problem is initially placed at the root of the tree. The original data is then split into two subproblems, one for each subtree. And so on. Eventually, the problem is solved at the leaves of the tree. A solution to the original problem is obtained by some mechanism that can reassemble the pieces. In fact, an implementation of the Chinese

Remainder Algorithm using balanced binary trees was first proposed by David Y. Y. Yun at the IBM T. J. Watson Research Center in Yorktown Heights, New York, in 1978. It served as the prototype for polymorphic algorithms in AXIOM.

In what follows, rather than perform a series of computations with a single expression, the expression is reduced modulo a number of integer primes, a computation is done with modular arithmetic for each prime, and the Chinese Remainder Algorithm is used to obtain the answer to the original problem. We illustrate this principle with the computation of $12^2 = 144$.

A list of moduli.

```
lm := [3,5,7,11]
```

$$[3, 5, 7, 11]$$

Type: List PositiveInteger

The expression `modTree(n, lm)` creates a balanced binary tree with leaf values `n mod m` for each modulus `m` in `lm`.

```
modTree(12,lm)
```

$$[0, 2, 5, 1]$$

Type: List Integer

Operation `modTree` does this using operations on balanced binary trees. We trace its steps. Create a balanced binary tree `t` of zeros with four leaves.

```
t := balancedBinaryTree(#lm, 0)
```

$$[[0, 0, 0], 0, [0, 0, 0]]$$

Type: BalancedBinaryTree NonNegativeInteger

The leaves of the tree are set to the individual moduli.

```
setleaves!(t,lm)
```

$$[[3, 0, 5], 0, [7, 0, 11]]$$

Type: BalancedBinaryTree NonNegativeInteger

Use `mapUp!` to do a bottom-up traversal of `t`, setting each interior node to the product of the values at the nodes of its children.

```
mapUp!(t,_*)
```

Type: PositiveInteger

The value at the node of every subtree is the product of the moduli of the leaves of the subtree.

```
t
```

$$[[3, 15, 5], 1155, [7, 77, 11]]$$

Type: BalancedBinaryTree NonNegativeInteger

Operation `mapDown!(t,a,fn)` replaces the value `v` at each node of `t` by `fn(a,v)`.

```
mapDown!(t,12,_rem)
```

$$[[0, 12, 2], 12, [5, 12, 1]]$$

Type: BalancedBinaryTree NonNegativeInteger

The operation `leaves` returns the leaves of the resulting tree. In this case, it returns the list of `12 mod m` for each modulus `m`.

```
leaves %
```

$$[0, 2, 5, 1]$$

Type: List NonNegativeInteger

Compute the square of the images of `12` modulo each `m`.

```
squares := [x**2 rem m for x in % for m in lm]
```

$$[0, 4, 4, 1]$$

Type: List NonNegativeInteger

Call the Chinese Remainder Algorithm to get the answer for 12^2.

```
chineseRemainder(%,lm)
```

$$144$$

Type: PositiveInteger

9.3 BinaryExpansion

All rational numbers have repeating binary expansions. Operations to access the individual bits of a binary expansion can be obtained by converting the value to `RadixExpansion(2)`. More examples of expansions are available in 9.14 on page 58, 9.29 on page 96, and 9.51 on page 184.

The expansion (of type `BinaryExpansion`) of a rational number is returned by the **binary** operation.

```
r := binary(22/7)
```

$$11.\overline{001}$$

`Type: BinaryExpansion`

Arithmetic is exact.

```
r + binary(6/7)
```

$$100$$

`Type: BinaryExpansion`

The period of the expansion can be short or long ...

```
[binary(1/i) for i in 102..106]
```

$$\left[0.0\overline{00000101}, 0.\overline{000000100111110001000101100101111001110010010101001}, \right.$$
$$0.000\overline{000100111011}, 0.\overline{000000100111},$$
$$\left. 0.0\overline{0000010011010100100001110011111011001010110111100011}\right]$$

`Type: List BinaryExpansion`

or very long.

```
binary(1/1007)
```

$$0.\overline{000000000100000010001010010010111100000111111000010111111001011000111110 1}$$
$$\overline{0001001110010011001100011001001010101111011010011000000001100001100111 10}$$
$$\overline{1110001101000101111010010001111011000010101110111001110101011100110010 10}$$
$$\overline{0101110000000011100011110010000000100100100110111001010100111010001101 1101}$$
$$\overline{1010111000100100000110010110110000000101100101111100010100000101010101 101}$$
$$\overline{0110000011011011101001010111111110101110101001100100001010011011000100 110}$$
$$\overline{00100010000100001100011101001111 0001}$$

`Type: BinaryExpansion`

These numbers are bona fide algebraic objects.

```
p := binary(1/4)*x**2 + binary(2/3)*x + binary(4/9)
```

$$0.01\ x^2 + 0.\overline{10}\ x + 0.\overline{011100}$$

Type: Polynomial BinaryExpansion

```
q := D(p, x)
```

$$0.1\ x + 0.\overline{10}$$

Type: Polynomial BinaryExpansion

```
g := gcd(p, q)
```

$$x + 1.\overline{01}$$

Type: Polynomial BinaryExpansion

9.4 BinarySearchTree

`BinarySearchTree(R)` is the domain of binary trees with elements of type `R`, ordered across the nodes of the tree. A non-empty binary search tree has a value of type `R`, and `right` and `left` binary search subtrees. If a subtree is empty, it is displayed as a period (".").

Define a list of values to be placed across the tree. The resulting tree has `8` at the root; all other elements are in the left subtree.

```
lv := [8,3,5,4,6,2,1,5,7]
```

$$[8, 3, 5, 4, 6, 2, 1, 5, 7]$$

Type: List PositiveInteger

A convenient way to create a binary search tree is to apply the operation `binarySearchTree` to a list of elements.

```
t := binarySearchTree lv
```

$$[[[1, 2, .], 3, [4, 5, [5, 6, 7]]], 8, .]$$

Type: BinarySearchTree PositiveInteger

Another approach is to first create an empty binary search tree of integers.

```
emptybst := empty()$BSTREE(INT)
```

$$[\,]$$

```
Type: BinarySearchTree Integer
```

Insert the value `8`. This establishes `8` as the root of the binary search tree. Values inserted later that are less than `8` get stored in the `left` subtree, others in the `right` subtree.

```
t1 := insert!(8,emptybst)
```

$$8$$

```
Type: BinarySearchTree Integer
```

Insert the value `3`. This number becomes the root of the `left` subtree of `t1`. For optimal retrieval, it is thus important to insert the middle elements first.

```
insert!(3,t1)
```

$$[3, 8, .]$$

```
Type: BinarySearchTree Integer
```

We go back to the original tree `t`. The leaves of the binary search tree are those which have empty `left` and `right` subtrees.

```
leaves t
```

$$[1, 4, 5, 7]$$

```
Type: List PositiveInteger
```

The operation `split(k,t)` returns a containing the two subtrees: one with all elements "less" than `k`, another with elements "greater" than `k`.

```
split(3,t)
```

$$[less = [1, 2, .], greater = [[., 3, [4, 5, [5, 6, 7]]], 8, .]]$$

```
Type: Record(less: BinarySearchTree PositiveInteger,greater:
                   BinarySearchTree PositiveInteger)
```

Define `insertRoot` to insert new elements by creating a new node.

```
insertRoot: (INT,BSTREE INT) -> BSTREE INT
```

```
Type: Void
```

The new node puts the inserted value between its “less” tree and “greater” tree.

```
insertRoot(x, t) ==
    a := split(x, t)
    node(a.less, x, a.greater)
```

Function `buildFromRoot` builds a binary search tree from a list of elements `ls` and the empty tree `emptybst`.

```
buildFromRoot ls == reduce(insertRoot,ls,emptybst)
```

Type: Void

Apply this to the reverse of the list `lv`.

```
rt := buildFromRoot reverse lv
```

$$[[[1, 2, .], 3, [4, 5, [5, 6, 7]]], 8, .]$$

Type: BinarySearchTree Integer

Have AXIOM check that these are equal.

```
(t = rt)@Boolean
```

true

Type: Boolean

9.5 CardinalNumber

The `CardinalNumber` domain can be used for values indicating the cardinality of sets, both finite and infinite. For example, the **dimension** operation in the category `VectorSpace` returns a cardinal number.

The non-negative integers have a natural construction as cardinals

```
0 = #{ }, 1 = {0}, 2 = {0, 1}, ..., n = {i | 0 <= i < n}.
```

The fact that `0` acts as a zero for the multiplication of cardinals is equivalent to the axiom of choice.

Cardinal numbers can be created by conversion from non-negative integers.

```
c0 := 0 :: CardinalNumber
```

$$0$$

Type: CardinalNumber

```
c1 := 1 :: CardinalNumber
```

$$1$$

Type: CardinalNumber

```
c2 := 2 :: CardinalNumber
```

$$2$$

Type: CardinalNumber

```
c3 := 3 :: CardinalNumber
```

$$3$$

Type: CardinalNumber

They can also be obtained as the named cardinal `Aleph(n)`.

```
A0 := Aleph 0
```

$$Aleph\,(0)$$

Type: CardinalNumber

```
A1 := Aleph 1
```

$$Aleph\,(1)$$

Type: CardinalNumber

The **finite?** operation tests whether a value is a finite cardinal, that is, a non-negative integer.

```
finite? c2
```

`true`

Type: Boolean

```
finite? A0
```

`false`

```
Type: Boolean
```

Similarly, the **countable?** operation determines whether a value is a countable cardinal, that is, finite or `Aleph(0)`.

```
countable? c2
```

true

```
Type: Boolean
```

```
countable? A0
```

true

```
Type: Boolean
```

```
countable? A1
```

false

```
Type: Boolean
```

Arithmetic operations are defined on cardinal numbers as follows: If `x = #X` and `y = #Y` then

$x + y = \#(X + Y)$	cardinality of the disjoint union
$x - y = \#(X - Y)$	cardinality of the relative complement
$x * y = \#(X * Y)$	cardinality of the Cartesian product
$x ** y = \#(X ** Y)$	cardinality of the set of maps from `Y` to `X`

Here are some arithmetic examples.

```
[c2 + c2, c2 + A1]
```

$$[4, Aleph\,(1)]$$

```
Type: List CardinalNumber
```

```
[c0*c2, c1*c2, c2*c2, c0*A1, c1*A1, c2*A1, A0*A1]
```

$$[0, 2, 4, 0, Aleph\,(1), Aleph\,(1), Aleph\,(1)]$$

```
Type: List CardinalNumber
```

```
[c2**c0, c2**c1, c2**c2, A1**c0, A1**c1, A1**c2]
```

$$[1, 2, 4, 1, Aleph\,(1), Aleph\,(1)]$$

```
Type: List CardinalNumber
```

Subtraction is a partial operation: it is not defined when subtracting a larger cardinal from a smaller one, nor when subtracting two equal infinite cardinals.

```
[c2-c1, c2-c2, c2-c3, A1-c2, A1-A0, A1-A1]
```

$$[1, 0, \texttt{"failed"}, Aleph\,(1), Aleph\,(1), \texttt{"failed"}]$$

```
Type: List Union(CardinalNumber,"failed")
```

The generalized continuum hypothesis asserts that

```
2**Aleph i = Aleph(i+1)
```

and is independent of the axioms of set theory.[1]

The `CardinalNumber` domain provides an operation to assert whether the hypothesis is to be assumed.

```
generalizedContinuumHypothesisAssumed true
```

true

When the generalized continuum hypothesis is assumed, exponentiation to a transfinite power is allowed.

```
[c0**A0, c1**A0, c2**A0, A0**A0, A0**A1, A1**A0, A1**A1]
```

$$[0, 1, Aleph\,(1), Aleph\,(1), Aleph\,(2), Aleph\,(1), Aleph\,(2)]$$

```
Type: List CardinalNumber
```

Three commonly encountered cardinal numbers are

$\mathtt{a} = \#\mathbf{Z}$	countable infinity
$\mathtt{c} = \#\mathbf{R}$	the continuum
$\mathtt{f} = \#\{\mathtt{g} \mid \mathtt{g} : [0,1] -> \mathbf{R}\}$	

In this domain, these values are obtained under the generalized continuum hypothesis in this way.

```
a := Aleph 0
```

$$Aleph\,(0)$$

```
Type: CardinalNumber
```

```
c := 2**a
```

[1] Goedel, *The consistency of the continuum hypothesis,* Ann. Math. Studies, Princeton Univ. Press, 1940.

$$Aleph\,(1)$$

Type: CardinalNumber

```
f := 2**c
```

$$Aleph\,(2)$$

Type: CardinalNumber

9.6 CartesianTensor

`CartesianTensor(i0,dim,R)` provides Cartesian tensors with components belonging to a commutative ring `R`. Tensors can be described as a generalization of vectors and matrices. This gives a concise *tensor algebra* for multilinear objects supported by the `CartesianTensor` domain. You can form the inner or outer product of any two tensors and you can add or subtract tensors with the same number of components. Additionally, various forms of traces and transpositions are useful.

The `CartesianTensor` constructor allows you to specify the minimum index for subscripting. In what follows we discuss in detail how to manipulate tensors.

Here we construct the domain of Cartesian tensors of dimension 2 over the integers, with indices starting at 1.

```
CT := CARTEN(i0 := 1, 2, Integer)
```

$$CartesianTensor(1, 2, Integer)$$

Type: Domain

Forming tensors

Scalars can be converted to tensors of rank zero.

```
t0:  CT := 8
```

$$8$$

Type: CartesianTensor(1,2,Integer)

```
rank t0
```

$$0$$

Type: NonNegativeInteger

Vectors (mathematical direct products, rather than one dimensional array structures) can be converted to tensors of rank one.

```
v: DirectProduct(2, Integer) := directProduct [3,4]
```

$$[3, 4]$$

Type: DirectProduct(2,Integer)

```
Tv: CT := v
```

$$[3, 4]$$

Type: CartesianTensor(1,2,Integer)

Matrices can be converted to tensors of rank two.

```
m: SquareMatrix(2, Integer) := matrix [ [1,2],[4,5] ]
```

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$$

Type: SquareMatrix(2,Integer)

```
Tm: CT := m
```

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$$

Type: CartesianTensor(1,2,Integer)

```
n: SquareMatrix(2, Integer) := matrix [ [2,3],[0,1] ]
```

$$\begin{bmatrix} 2 & 3 \\ 0 & 1 \end{bmatrix}$$

Type: SquareMatrix(2,Integer)

```
Tn: CT := n
```

$$\begin{bmatrix} 2 & 3 \\ 0 & 1 \end{bmatrix}$$

Type: CartesianTensor(1,2,Integer)

In general, a tensor of rank `k` can be formed by making a list of rank `k-1` tensors or, alternatively, a `k`-deep nested list of lists.

```
t1:  CT := [2, 3]
```

$$[2,3]$$

Type: CartesianTensor(1,2,Integer)

```
rank t1
```

$$1$$

Type: PositiveInteger

```
t2:  CT := [t1, t1]
```

$$\left[\begin{array}{cc} 2 & 3 \\ 2 & 3 \end{array}\right]$$

Type: CartesianTensor(1,2,Integer)

```
t3:  CT := [t2, t2]
```

$$\left[\left[\begin{array}{cc} 2 & 3 \\ 2 & 3 \end{array}\right], \left[\begin{array}{cc} 2 & 3 \\ 2 & 3 \end{array}\right]\right]$$

Type: CartesianTensor(1,2,Integer)

```
tt:  CT := [t3, t3]; tt := [tt, tt]
```

$$\left[\left[\begin{array}{cc} \left[\begin{array}{cc} 2 & 3 \\ 2 & 3 \\ 2 & 3 \\ 2 & 3 \end{array}\right] & \left[\begin{array}{cc} 2 & 3 \\ 2 & 3 \\ 2 & 3 \\ 2 & 3 \end{array}\right] \end{array}\right], \left[\begin{array}{cc} \left[\begin{array}{cc} 2 & 3 \\ 2 & 3 \\ 2 & 3 \\ 2 & 3 \end{array}\right] & \left[\begin{array}{cc} 2 & 3 \\ 2 & 3 \\ 2 & 3 \\ 2 & 3 \end{array}\right] \end{array}\right]\right]$$

Type: CartesianTensor(1,2,Integer)

```
rank tt
```

$$5$$

Type: PositiveInteger

Multiplication

Given two tensors of rank `k1` and `k2`, the outer **product** forms a new tensor of rank `k1+k2`. Here

$$T_{mn}(i,j,k,l) = T_m(i,j)\ T_n(k,l)$$

```
Tmn := product(Tm, Tn)
```

$$\left[\begin{array}{cc} \left[\begin{array}{cc} 2 & 3 \\ 0 & 1 \end{array}\right] & \left[\begin{array}{cc} 4 & 6 \\ 0 & 2 \end{array}\right] \\ \left[\begin{array}{cc} 8 & 12 \\ 0 & 4 \end{array}\right] & \left[\begin{array}{cc} 10 & 15 \\ 0 & 5 \end{array}\right] \end{array}\right]$$

`Type: CartesianTensor(1,2,Integer)`

The inner product (**contract**) forms a tensor of rank `k1+k2-2`. This product generalizes the vector dot product and matrix-vector product by summing component products along two indices.

Here we sum along the second index of T_m and the first index of T_v. Here

$$T_{mv} = \sum_{j=1}^{\dim} T_m(i,j)\ T_v(j)$$

```
Tmv := contract(Tm,2,Tv,1)
```

$$[11, 32]$$

`Type: CartesianTensor(1,2,Integer)`

The multiplication operator "`*`" is scalar multiplication or an inner product depending on the ranks of the arguments.

If either argument is rank zero it is treated as scalar multiplication. Otherwise, `a*b` is the inner product summing the last index of `a` with the first index of `b`.

```
Tm*Tv
```

$$[11, 32]$$

`Type: CartesianTensor(1,2,Integer)`

This definition is consistent with the inner product on matrices and vectors.

```
Tmv = m * v
```

$$[11, 32] = [11, 32]$$

`Type: Equation CartesianTensor(1,2,Integer)`

Selecting Components

For tensors of low rank (that is, four or less), components can be selected by applying the tensor to its indices.

```
t0()
```

$$8$$

Type: PositiveInteger

```
t1(1+1)
```

$$3$$

Type: PositiveInteger

```
t2(2,1)
```

$$2$$

Type: PositiveInteger

```
t3(2,1,2)
```

$$3$$

Type: PositiveInteger

```
Tmn(2,1,2,1)
```

$$0$$

Type: NonNegativeInteger

A general indexing mechanism is provided for a list of indices.

```
t0[]
```

$$8$$

Type: PositiveInteger

```
t1[2]
```

$$3$$

Type: PositiveInteger

```
t2[2,1]
```

$$2$$

Type: PositiveInteger

The general mechanism works for tensors of arbitrary rank, but is somewhat less efficient since the intermediate index list must be created.

```
t3[2,1,2]
```

$$3$$

Type: PositiveInteger

```
Tmn[2,1,2,1]
```

$$0$$

Type: NonNegativeInteger

Contraction

A "contraction" between two tensors is an inner product, as we have seen above. You can also contract a pair of indices of a single tensor. This corresponds to a "trace" in linear algebra. The expression `contract(t,k1,k2)` forms a new tensor by summing the diagonal given by indices in position `k1` and `k2`.

This is the tensor given by

$$xT_{mn} = \sum_{k=1}^{\dim} T_{mn}(k,k,i,j)$$

```
cTmn := contract(Tmn,1,2)
```

$$\begin{bmatrix} 12 & 18 \\ 0 & 6 \end{bmatrix}$$

Type: CartesianTensor(1,2,Integer)

Since `Tmn` is the outer product of matrix `m` and matrix `n`, the above is equivalent to this.

```
trace(m) * n
```

$$\begin{bmatrix} 12 & 18 \\ 0 & 6 \end{bmatrix}$$

Type: SquareMatrix(2,Integer)

In this and the next few examples, we show all possible contractions of `Tmn` and their matrix algebra equivalents.

```
contract(Tmn,1,2) = trace(m) * n
```

$$\begin{bmatrix} 12 & 18 \\ 0 & 6 \end{bmatrix} = \begin{bmatrix} 12 & 18 \\ 0 & 6 \end{bmatrix}$$

Type: Equation CartesianTensor(1,2,Integer)

```
contract(Tmn,1,3) = transpose(m) * n
```

$$\begin{bmatrix} 2 & 7 \\ 4 & 11 \end{bmatrix} = \begin{bmatrix} 2 & 7 \\ 4 & 11 \end{bmatrix}$$

Type: Equation CartesianTensor(1,2,Integer)

```
contract(Tmn,1,4) = transpose(m) * transpose(n)
```

$$\begin{bmatrix} 14 & 4 \\ 19 & 5 \end{bmatrix} = \begin{bmatrix} 14 & 4 \\ 19 & 5 \end{bmatrix}$$

Type: Equation CartesianTensor(1,2,Integer)

```
contract(Tmn,2,3) = m * n
```

$$\begin{bmatrix} 2 & 5 \\ 8 & 17 \end{bmatrix} = \begin{bmatrix} 2 & 5 \\ 8 & 17 \end{bmatrix}$$

Type: Equation CartesianTensor(1,2,Integer)

```
contract(Tmn,2,4) = m * transpose(n)
```

$$\begin{bmatrix} 8 & 2 \\ 23 & 5 \end{bmatrix} = \begin{bmatrix} 8 & 2 \\ 23 & 5 \end{bmatrix}$$

Type: Equation CartesianTensor(1,2,Integer)

```
contract(Tmn,3,4) = trace(n) * m
```

$$\begin{bmatrix} 3 & 6 \\ 12 & 15 \end{bmatrix} = \begin{bmatrix} 3 & 6 \\ 12 & 15 \end{bmatrix}$$

Type: Equation CartesianTensor(1,2,Integer)

Transpositions

You can exchange any desired pair of indices using the **transpose** operation.

Here the indices in positions one and three are exchanged, that is,

$$tT_{mn}(i,j,k,l) = T_{mn}(k,j,i,l).$$

```
tTmn := transpose(Tmn,1,3)
```

$$\left[\begin{array}{cc} \left[\begin{array}{cc} 2 & 3 \\ 8 & 12 \end{array}\right] & \left[\begin{array}{cc} 4 & 6 \\ 10 & 15 \end{array}\right] \\ \left[\begin{array}{cc} 0 & 1 \\ 0 & 4 \end{array}\right] & \left[\begin{array}{cc} 0 & 2 \\ 0 & 5 \end{array}\right] \end{array}\right]$$

Type: `CartesianTensor(1,2,Integer)`

If no indices are specified, the first and last index are exchanged.

```
transpose Tmn
```

$$\left[\begin{array}{cc} \left[\begin{array}{cc} 2 & 8 \\ 0 & 0 \end{array}\right] & \left[\begin{array}{cc} 4 & 10 \\ 0 & 0 \end{array}\right] \\ \left[\begin{array}{cc} 3 & 12 \\ 1 & 4 \end{array}\right] & \left[\begin{array}{cc} 6 & 15 \\ 2 & 5 \end{array}\right] \end{array}\right]$$

Type: `CartesianTensor(1,2,Integer)`

This is consistent with the matrix transpose.

```
transpose Tm = transpose m
```

$$\left[\begin{array}{cc} 1 & 4 \\ 2 & 5 \end{array}\right] = \left[\begin{array}{cc} 1 & 4 \\ 2 & 5 \end{array}\right]$$

Type: `Equation CartesianTensor(1,2,Integer)`

If a more complicated reordering of the indices is required, then the **reindex** operation can be used. This operation allows the indices to be arbitrarily permuted.

This defines $rT_{mn}(i,j,k,l) = T_{mn}(i,l,j,k)$.

```
rTmn := reindex(Tmn, [1,4,2,3])
```

$$\left[\begin{array}{cc} \left[\begin{array}{cc} 2 & 0 \\ 4 & 0 \end{array}\right] & \left[\begin{array}{cc} 3 & 1 \\ 6 & 2 \end{array}\right] \\ \left[\begin{array}{cc} 8 & 0 \\ 10 & 0 \end{array}\right] & \left[\begin{array}{cc} 12 & 4 \\ 15 & 5 \end{array}\right] \end{array}\right]$$

Type: `CartesianTensor(1,2,Integer)`

Arithmetic

Tensors of equal rank can be added or subtracted so arithmetic expressions can be used to produce new tensors.

```
tt := transpose(Tm)*Tn - Tn*transpose(Tm)
```

$$\begin{bmatrix} -6 & -16 \\ 2 & 6 \end{bmatrix}$$

Type: `CartesianTensor(1,2,Integer)`

```
Tv*(tt+Tn)
```

$$[-4, -11]$$

Type: `CartesianTensor(1,2,Integer)`

```
reindex(product(Tn,Tn),[4,3,2,1])+3*Tn*product(Tm,Tm)
```

$$\begin{bmatrix} \begin{bmatrix} 46 & 84 \\ 174 & 212 \end{bmatrix} & \begin{bmatrix} 57 & 114 \\ 228 & 285 \end{bmatrix} \\ \begin{bmatrix} 18 & 24 \\ 57 & 63 \end{bmatrix} & \begin{bmatrix} 17 & 30 \\ 63 & 76 \end{bmatrix} \end{bmatrix}$$

Type: `CartesianTensor(1,2,Integer)`

Specific Tensors

Two specific tensors have properties which depend only on the dimension.

The Kronecker delta satisfies

$$delta(i,j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

```
delta:  CT := kroneckerDelta()
```

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Type: `CartesianTensor(1,2,Integer)`

This can be used to reindex via contraction.

```
contract(Tmn, 2, delta, 1) = reindex(Tmn, [1,3,4,2])
```

$$\begin{bmatrix} \begin{bmatrix} 2 & 4 \\ 3 & 6 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 1 & 2 \end{bmatrix} \\ \begin{bmatrix} 8 & 10 \\ 12 & 15 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 4 & 5 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} 2 & 4 \\ 3 & 6 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 1 & 2 \end{bmatrix} \\ \begin{bmatrix} 8 & 10 \\ 12 & 15 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 4 & 5 \end{bmatrix} \end{bmatrix}$$

Type: Equation CartesianTensor(1,2,Integer)

The Levi Civita symbol determines the sign of a permutation of indices.

```
epsilon:CT := leviCivitaSymbol()
```

$$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

Type: CartesianTensor(1,2,Integer)

Here we have:

$$\text{epsilon}(i_1, \ldots, i_{dim}) = \begin{cases} +1 & \text{if } i_1, \ldots, i_{dim} \text{ is an even permutation of} \\ & i_0, \ldots, i_0 + \text{dim} - 1 \\ -1 & \text{if } i_1, \ldots, i_{dim} \text{ is an odd permutation of} \\ & i_0, \ldots, i_0 + \text{dim} - 1 \\ 0 & \text{if } i_1, \ldots, i_{dim} \text{ is } \textit{not} \text{ a permutation of} \\ & i_0, \ldots, i_0 + \text{dim} - 1 \end{cases}$$

This property can be used to form determinants.

```
contract(epsilon*Tm*epsilon, 1,2) = 2 * determinant m
```

$$-6 = -6$$

Type: Equation CartesianTensor(1,2,Integer)

Properties of the CartesianTensor domain

`GradedModule(R,E)` denotes "`E`-graded `R`-module", that is, a collection of `R`-modules indexed by an abelian monoid `E`. An element `g` of `G[s]` for some specific `s` in `E` is said to be an element of `G` with **degree** `s`. Sums are defined in each module `G[s]` so two elements of `G` can be added if they have the same degree. Morphisms can be defined and composed by degree to give the mathematical category of graded modules.

`GradedAlgebra(R,E)` denotes "`E`-graded `R`-algebra." A graded algebra is a graded module together with a degree preserving `R`-bilinear map, called the **product**.

```
degree(product(a,b))    = degree(a) + degree(b)

product(r*a,b)          = product(a,r*b) = r*product(a,b)
product(a1+a2,b)        = product(a1,b) + product(a2,b)
product(a,b1+b2)        = product(a,b1) + product(a,b2)
product(a,product(b,c)) = product(product(a,b),c)
```

The domain CartesianTensor(i0, dim, R) belongs to the category Graded Algebra (R, NonNegativeInteger). The non-negative integer **degree** is the tensor rank and the graded algebra **product** is the tensor outer product. The graded module addition captures the notion that only tensors of equal rank can be added.

If V is a vector space of dimension dim over R, then the tensor module T[k](V) is defined as

```
T[0](V) = R
T[k](V) = T[k-1](V) * V
```

where "*" denotes the R-module tensor **product**. CartesianTensor(i0,dim,R) is the graded algebra in which the degree k module is T[k](V).

Tensor Calculus

It should be noted here that often tensors are used in the context of tensor-valued manifold maps. This leads to the notion of covariant and contravariant bases with tensor component functions transforming in specific ways under a change of coordinates on the manifold. This is no more directly supported by the CartesianTensor domain than it is by the Vector domain. However, it is possible to have the components implicitly represent component maps by choosing a polynomial or expression type for the components. In this case, it is up to the user to satisfy any constraints which arise on the basis of this interpretation.

9.7 Character

The members of the domain Character are values representing letters, numerals and other text elements. For more information on related topics, see 9.8 on page 25 and 9.61 on page 205.

Characters can be obtained using String notation.

```
chars := [char "a", char "A", char "X", char "8", char "+"]
```

$$[a, A, X, 8, +]$$

Type: List Character

Certain characters are available by name. This is the blank character.

```
space()
```

Type: Character

This is the quote that is used in strings.

```
quote()
```

"

Type: Character

This is the escape character that allows quotes and other characters within strings.

```
escape()
```

_

Type: Character

Characters are represented as integers in a machine-dependent way. The integer value can be obtained using the **ord** operation. It is always true that `char(ord c) = c` and `ord(char i) = i`, provided that `i` is in the range `0..size()$Character-1`.

```
[ord c for c in chars]
```

$$[97, 65, 88, 56, 43]$$

Type: List Integer

The **lowerCase** operation converts an upper case letter to the corresponding lower case letter. If the argument is not an upper case letter, then it is returned unchanged.

```
[upperCase c for c in chars]
```

$$[A, A, X, 8, +]$$

Type: List Character

Likewise, the **upperCase** operation converts lower case letters to upper case.

```
[lowerCase c for c in chars]
```

$$[a, a, x, 8, +]$$

Type: List Character

A number of tests are available to determine whether characters belong to certain families.

```
[alphabetic? c for c in chars]
```

$$[\texttt{true}, \texttt{true}, \texttt{true}, \texttt{false}, \texttt{false}]$$

Type: List Boolean

```
[upperCase? c for c in chars]
```

$$[\texttt{false}, \texttt{true}, \texttt{true}, \texttt{false}, \texttt{false}]$$

Type: List Boolean

```
[lowerCase? c for c in chars]
```

$$[\texttt{true}, \texttt{false}, \texttt{false}, \texttt{false}, \texttt{false}]$$

Type: List Boolean

```
[digit? c for c in chars]
```

$$[\texttt{false}, \texttt{false}, \texttt{false}, \texttt{true}, \texttt{false}]$$

Type: List Boolean

```
[hexDigit? c for c in chars]
```

$$[\texttt{true}, \texttt{true}, \texttt{false}, \texttt{true}, \texttt{false}]$$

Type: List Boolean

```
[alphanumeric? c for c in chars]
```

$$[\texttt{true}, \texttt{true}, \texttt{true}, \texttt{true}, \texttt{false}]$$

Type: List Boolean

9.8 CharacterClass

The `CharacterClass` domain allows classes of characters to be defined and manipulated efficiently.

Character classes can be created by giving either a string or a list of characters.

```
cl1 := charClass [char "a", char "e", char "i", char "o", char
"u", char "y"]
```

"aeiouy"

Type: CharacterClass

```
cl2 := charClass "bcdfghjklmnpqrstvwxyz"
```

"bcdfghjklmnpqrstvwxyz"

Type: CharacterClass

A number of character classes are predefined for convenience.

```
digit()
```

"0123456789"

Type: CharacterClass

```
hexDigit()
```

"0123456789ABCDEFabcdef"

Type: CharacterClass

```
upperCase()
```

"ABCDEFGHIJKLMNOPQRSTUVWXYZ"

Type: CharacterClass

```
lowerCase()
```

"abcdefghijklmnopqrstuvwxyz"

Type: CharacterClass

```
alphabetic()
```

"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

Type: CharacterClass

```
alphanumeric()
```

"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"

Type: CharacterClass

You can quickly test whether a character belongs to a class.

```
member?(char "a", cl1)
```

true

Type: Boolean

```
member?(char "a", cl2)
```

false

Type: Boolean

Classes have the usual set operations because the `CharacterClass` domain belongs to the category `FiniteSetAggregate(Character)`.

```
intersect(cl1, cl2)
```

"y"

Type: CharacterClass

```
union(cl1,cl2)
```

"abcdefghijklmnopqrstuvwxyz"

Type: CharacterClass

```
difference(cl1,cl2)
```

"aeiou"

Type: CharacterClass

```
intersect(complement(cl1),cl2)
```

"bcdfghjklmnpqrstvwxz"

Type: CharacterClass

You can modify character classes by adding or removing characters.

```
insert!(char "a", cl2)
```

"abcdfghjklmnpqrstvwxyz"

Type: CharacterClass

```
remove!(char "b", cl2)
```

"acdfghjklmnpqrstvwxyz"

Type: CharacterClass

For more information on related topics, see 9.7 on page 23 and 9.61 on page 205.

9.9 CliffordAlgebra

`CliffordAlgebra(n,K,Q)` defines a vector space of dimension 2^n over the field K with a given quadratic form `Q`. If $\{e_1, \ldots, e_n\}$ is a basis for K^n then

$$\begin{array}{rcl} \{ & 1 & \\ & e_i & \text{for } 1 \le i \le n \\ & e_{i_1}, e_{i_2} & \text{for } 1 \le i_1 \le i_2 \le n \\ & e_1\ e_2\ \cdots\ e_n & \} \end{array}$$

is a basis for the Clifford algebra. The algebra is defined by the relations

$$\begin{array}{ll} e_i\ e_i & = Q(e_i) \\ e_i\ e_j & = -e_j\ e_i,\ i \ne j \end{array}$$

Examples of Clifford Algebras are gaussians (complex numbers), quaternions, exterior algebras and spin algebras.

9.9.1 The Complex Numbers as a Clifford Algebra

This is the field over which we will work, rational functions with integer coefficients.

```
K := Fraction Polynomial Integer
```

Fraction Polynomial Integer

`Type: Domain`

We use this matrix for the quadratic form.

```
m := matrix [ [-1] ]
```

$$\begin{bmatrix} -1 \end{bmatrix}$$

`Type: Matrix Integer`

We get complex arithmetic by using this domain.

```
C := CliffordAlgebra(1, K, quadraticForm m)
```

CliffordAlgebra(1,Fraction Polynomial Integer,MATRIX)

`Type: Domain`

Here is `i`, the usual square root of `-1`.

```
i:  C := e(1)
```

$$e_1$$

Type: CliffordAlgebra(1,Fraction Polynomial Integer,MATRIX)

Here are some examples of the arithmetic.

```
x := a + b * i
```

$$a + b\ e_1$$

Type: CliffordAlgebra(1,Fraction Polynomial Integer,MATRIX)

```
y := c + d * i
```

$$c + d\ e_1$$

Type: CliffordAlgebra(1,Fraction Polynomial Integer,MATRIX)

See 9.10 on page 34 for examples of AXIOM's constructor implementing complex numbers.

```
x * y
```

$$-b\ d + a\ c + (a\ d + b\ c)\ e_1$$

Type: CliffordAlgebra(1,Fraction Polynomial Integer,MATRIX)

9.9.2 The Quaternion Numbers as a Clifford Algebra

```
K := Fraction Polynomial Integer
```

Fraction Polynomial Integer

Type: Domain

We use this matrix for the quadratic form.

```
m := matrix [ [-1,0],[0,-1] ]
```

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$$

Type: Matrix Integer

The resulting domain is the quaternions.

```
H := CliffordAlgebra(2, K, quadraticForm m)
```

CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

Type: Domain

We use Hamilton's notation for i,j,k.

```
i: H := e(1)
```

$$e_1$$

Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

```
j: H := e(2)
```

$$e_2$$

Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

```
k: H := i * j
```

$$e_1\ e_2$$

Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

```
x := a + b * i + c * j + d * k
```

$$a + b\ e_1 + c\ e_2 + d\ e_1\ e_2$$

Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

```
y := e + f * i + g * j + h * k
```

$$e + f\ e_1 + g\ e_2 + h\ e_1\ e_2$$

Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

```
x + y
```

$$e + a + (f + b)\ e_1 + (g + c)\ e_2 + (h + d)\ e_1\ e_2$$

Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

```
x * y
```

$$-d\ h - c\ g - b\ f + a\ e + (c\ h - d\ g + a\ f + b\ e)\ e_1 +$$

$$(-b\ h + a\ g + d\ f + c\ e)\ e_2 + (a\ h + b\ g - c\ f + d\ e)\ e_1\ e_2$$

Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

See 9.50 on page 182 for examples of AXIOM's constructor implementing quaternions.

```
y * x
```

$$-d\ h - c\ g - b\ f + a\ e + (-c\ h + d\ g + a\ f + b\ e)\ e_1 +$$

$$(b\ h + a\ g - d\ f + c\ e)\ e_2 + (a\ h - b\ g + c\ f + d\ e)\ e_1\ e_2$$

Type: CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

9.9.3 The Exterior Algebra on a Three Space

This is the field over which we will work, rational functions with integer coefficients.

```
K := Fraction Polynomial Integer
```

Fraction Polynomial Integer

Type: Domain

If we chose the three by three zero quadratic form, we obtain the exterior algebra on `e(1),e(2),e(3)`.

```
Ext := CliffordAlgebra(3, K, quadraticForm 0)
```

CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

Type: Domain

This is a three dimensional vector algebra. We define `i`, `j`, `k` as the unit vectors.

```
i: Ext := e(1)
```

$$e_1$$

Type: CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

```
j: Ext := e(2)
```

$$e_2$$

Type: CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

```
k:  Ext := e(3)
```

$$e_3$$

Type: CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

Now it is possible to do arithmetic.

```
x := x1*i + x2*j + x3*k
```

$$x1\ e_1 + x2\ e_2 + x3\ e_3$$

Type: CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

```
y := y1*i + y2*j + y3*k
```

$$y1\ e_1 + y2\ e_2 + y3\ e_3$$

Type: CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

```
x + y
```

$$(y1 + x1)\ e_1 + (y2 + x2)\ e_2 + (y3 + x3)\ e_3$$

Type: CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

```
x * y + y * x
```

$$0$$

Type: CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

On an `n` space, a grade `p` form has a dual `n-p` form. In particular, in three space the dual of a grade two element identifies `e1*e2->e3, e2*e3->e1, e3*e1->e2`.

```
dual2 a == coefficient(a,[2,3]) * i + coefficient(a,[3,1]) * j +
coefficient(a,[1,2]) * k
```

Type: Void

The vector cross product is then given by this.

```
dual2(x*y)
```

$$(x2\ y3 - x3\ y2)\ e_1 + (-x1\ y3 + x3\ y1)\ e_2 + (x1\ y2 - x2\ y1)\ e_3$$

Type: CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

9.9.4 The Dirac Spin Algebra

In this section we will work over the field of rational numbers.

```
K := Fraction Integer
```

Fraction Integer

Type: Domain

We define the quadratic form to be the Minkowski space-time metric.

```
g := matrix [ [1,0,0,0], [0,-1,0,0], [0,0,-1,0], [0,0,0,-1] ]
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

Type: Matrix Integer

We obtain the Dirac spin algebra used in Relativistic Quantum Field Theory.

```
D := CliffordAlgebra(4,K, quadraticForm g)
```

CliffordAlgebra(4,Fraction Integer,MATRIX)

Type: Domain

The usual notation for the basis is γ with a superscript. For AXIOM input we will use `gam(i)`:

```
gam := [e(i)$D for i in 1..4]
```

$$[e_1, e_2, e_3, e_4]$$

Type: List CliffordAlgebra(4,Fraction Integer,MATRIX)

There are various contraction identities of the form

```
g(l,t)*gam(l)*gam(m)*gam(n)*gam(r)*gam(s)*gam(t) =
      2*(gam(s)gam(m)gam(n)gam(r) + gam(r)*gam(n)*gam(m)*gam(s))
```

where a sum over `l` and `t` is implied.

Verify this identity for particular values of `m,n,r,s`.

```
m := 1; n:= 2; r := 3; s := 4;
```

Type: PositiveInteger

```
lhs := reduce(+, [reduce(+, [
g(l,t)*gam(l)*gam(m)*gam(n)*gam(r)*gam(s)*gam(t) for l in 1..4])
for t in 1..4])
```

$$-4\ e_1\ e_2\ e_3\ e_4$$

Type: CliffordAlgebra(4,Fraction Integer,MATRIX)

```
rhs := 2*(gam s * gam m*gam n*gam r + gam r*gam n*gam m*gam s)
```

$$-4\ e_1\ e_2\ e_3\ e_4$$

Type: CliffordAlgebra(4,Fraction Integer,MATRIX)

9.10 Complex

The `Complex` constructor implements complex objects over a commutative ring `R`. Typically, the ring `R` is `Integer`, `Fraction Integer`, `Float` or `DoubleFloat`. `R` can also be a symbolic type, like `Polynomial Integer`.

Complex objects are created by the **complex** operation.

```
a := complex(4/3,5/2)
```

$$\frac{4}{3}+\frac{5}{2}\ \%i$$

Type: Complex Fraction Integer

```
b := complex(4/3,-5/2)
```

$$\frac{4}{3}-\frac{5}{2}\ \%i$$

Type: Complex Fraction Integer

The standard arithmetic operations are available.

```
a + b
```

$$\frac{8}{3}$$

Type: Complex Fraction Integer

```
a - b
```

$$5\ \%i$$

Type: Complex Fraction Integer

```
a * b
```

$$\frac{289}{36}$$

Type: Complex Fraction Integer

If R is a field, you can also divide the complex objects.

```
a / b
```

$$-\frac{161}{289}+\frac{240}{289}\ \%i$$

Type: Complex Fraction Integer

Use a conversion to view the last object as a fraction of complex integers.

```
% :: Fraction Complex Integer
```

$$\frac{-15+8\ \%i}{15+8\ \%i}$$

Type: Fraction Complex Integer

The predefined macro `%i` is defined to be `complex(0,1)`.

```
3.4 + 6.7 * %i
```

$$3.4+6.7\ \%i$$

Type: Complex Float

You can also compute the **conjugate** and **norm** of a complex number.

```
conjugate a
```

$$\frac{4}{3}-\frac{5}{2}\ \%i$$

Type: Complex Fraction Integer

```
norm a
```

$$\frac{289}{36}$$

Type: Fraction Integer

The **real** and **imag** operations are provided to extract the real and imaginary parts, respectively.

```
real a
```

$$\frac{4}{3}$$

Type: Fraction Integer

```
imag a
```

$$\frac{5}{2}$$

Type: Fraction Integer

The domain `Complex Integer` is also called the Gaussian integers. If `R` is the integers (or, more generally, a `EuclideanDomain`), you can compute greatest common divisors.

```
gcd(13 - 13*%i,31 + 27*%i)
```

$$5 + \%i$$

Type: Complex Integer

You can also compute least common multiples.

```
lcm(13 - 13*%i,31 + 27*%i)
```

$$143 - 39\ \%i$$

Type: Complex Integer

You can **factor** Gaussian integers.

```
factor(13 - 13*%i)
```

$$-(1 + \%i)\ (2 + 3\ \%i)\ (3 + 2\ \%i)$$

Type: Factored Complex Integer

```
factor complex(2,0)
```

$$-i\ (1 + \%i)^2$$

Type: Factored Complex Integer

9.11 ContinuedFraction

Continued fractions have been a fascinating and useful tool in mathematics for well over three hundred years. AXIOM implements continued fractions for fractions of any Euclidean domain. In practice, this usually means rational numbers. In this section we demonstrate some of the operations available for manipulating both finite and infinite continued fractions. It may be helpful if you review 9.60 on page 202 to remind yourself of some of the operations with streams.

The `ContinuedFraction` domain is a field and therefore you can add, subtract, multiply and divide the fractions.

The **continuedFraction** operation converts its fractional argument to a continued fraction.

```
c := continuedFraction(314159/100000)
```

$$3 + \frac{1|}{|7} + \frac{1|}{|15} + \frac{1|}{|1} + \frac{1|}{|25} + \frac{1|}{|1} + \frac{1|}{|7} + \frac{1|}{|4}$$

Type: `ContinuedFraction Integer`

This display is a compact form of the bulkier

$$3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1 + \cfrac{1}{25 + \cfrac{1}{1 + \cfrac{1}{7 + \cfrac{1}{4}}}}}}}$$

You can write any rational number in a similar form. The fraction will be finite and you can always take the "numerators" to be `1`. That is, any rational number can be written as a simple, finite continued fraction of the form

$$a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots\, a_{n-1} + \cfrac{1}{a_n}}}}$$

The a_i are called partial quotients and the operation **partialQuotients** creates a stream of them.

```
partialQuotients c
```

$$[3, 7, 15, 1, 25, 1, 7, \ldots]$$

Type: Stream Integer

By considering more and more of the fraction, you get the **convergents**. For example, the first convergent is a_1, the second is $a_1 + 1/a_2$ and so on.

```
convergents c
```

$$\left[3, \frac{22}{7}, \frac{333}{106}, \frac{355}{113}, \frac{9208}{2931}, \frac{9563}{3044}, \frac{76149}{24239}, \ldots\right]$$

Type: Stream Fraction Integer

Since this is a finite continued fraction, the last convergent is the original rational number, in reduced form. The result of **approximants** is always an infinite stream, though it may just repeat the "last" value.

```
approximants c
```

$$\left[3, \frac{22}{7}, \frac{333}{106}, \frac{355}{113}, \frac{9208}{2931}, \frac{9563}{3044}, \frac{76149}{24239}, \ldots\right]$$

Type: Stream Fraction Integer

Inverting `c` only changes the partial quotients of its fraction by inserting a `0` at the beginning of the list.

```
pq := partialQuotients(1/c)
```

$$[0, 3, 7, 15, 1, 25, 1, \ldots]$$

Type: Stream Integer

Do this to recover the original continued fraction from this list of partial quotients. The three-argument form of the **continuedFraction** operation takes an element which is the whole part of the fraction, a stream of elements which are the numerators of the fraction, and a stream of elements which are the denominators of the fraction.

```
continuedFraction(first pq,repeating [1],rest pq)
```

$$\frac{1|}{|3} + \frac{1|}{|7} + \frac{1|}{|15} + \frac{1|}{|1} + \frac{1|}{|25} + \frac{1|}{|1} + \frac{1|}{|7} + \ldots$$

Type: ContinuedFraction Integer

The streams need not be finite for **continuedFraction**. Can you guess which irrational number has the following continued fraction? See the end of this section for the answer.

```
z:=continuedFraction(3,repeating [1],repeating [3,6])
```

$$3 + \frac{1|}{|3} + \frac{1|}{|6} + \frac{1|}{|3} + \frac{1|}{|6} + \frac{1|}{|3} + \frac{1|}{|6} + \frac{1|}{|3} + \ldots$$

Type: ContinuedFraction Integer

In 1737 Euler discovered the infinite continued fraction expansion

$$\frac{e-1}{2} = \cfrac{1}{1 + \cfrac{1}{6 + \cfrac{1}{10 + \cfrac{1}{14 + \cdots}}}}$$

We use this expansion to compute rational and floating point approximations of `e`.[2]

By looking at the above expansion, we see that the whole part is `0` and the numerators are all equal to `1`. This constructs the stream of denominators.

```
dens:Stream Integer := cons(1,generate((x+->x+4),6))
```

$$[1, 6, 10, 14, 18, 22, 26, \ldots]$$

Type: Stream Integer

Therefore this is the continued fraction expansion for $(e-1)/2$.

```
cf := continuedFraction(0,repeating [1],dens)
```

$$\frac{1|}{|1} + \frac{1|}{|6} + \frac{1|}{|10} + \frac{1|}{|14} + \frac{1|}{|18} + \frac{1|}{|22} + \frac{1|}{|26} + \ldots$$

Type: ContinuedFraction Integer

These are the rational number convergents.

```
ccf := convergents cf
```

$$\left[0, 1, \frac{6}{7}, \frac{61}{71}, \frac{860}{1001}, \frac{15541}{18089}, \frac{342762}{398959}, \ldots\right]$$

Type: Stream Fraction Integer

You can get rational convergents for `e` by multiplying by `2` and adding `1`.

```
eConvergents := [2*e + 1 for e in ccf]
```

[2]For this and other interesting expansions, see C. D. Olds, *Continued Fractions,* New Mathematical Library, (New York: Random House, 1963), pp. 134–139.

$$\left[1, 3, \frac{19}{7}, \frac{193}{71}, \frac{2721}{1001}, \frac{49171}{18089}, \frac{1084483}{398959}, \ldots\right]$$

Type: `Stream Fraction Integer`

You can also compute the floating point approximations to these convergents.

```
eConvergents :: Stream Float
```

$$[1.0, 3.0, 2.7142857142857142857, 2.7183098591549295775,$$
$$2.7182817182817182817, 2.7182818287356957267,$$
$$2.7182818284\ 585634113, \ldots]$$

Type: `Stream Float`

Compare this to the value of `e` computed by the **exp** operation in `Float`.

```
exp 1.0
```

$$2.7182818284\ 590452354$$

Type: `Float`

In about 1658, Lord Brouncker established the following expansion for $4/\pi$,

$$1 + \cfrac{1}{2 + \cfrac{9}{2 + \cfrac{25}{2 + \cfrac{49}{2 + \cfrac{81}{2 + \cdots}}}}}$$

Let's use this expansion to compute rational and floating point approximations for π.

```
cf := continuedFraction(1,[(2*i+1)**2 for i in 0..],repeating
[2])
```

$$1 + \frac{1|}{|2} + \frac{9|}{|2} + \frac{25|}{|2} + \frac{49|}{|2} + \frac{81|}{|2} + \frac{121|}{|2} + \frac{169|}{|2} + \ldots$$

Type: `ContinuedFraction Integer`

```
ccf := convergents cf
```

$$\left[1, \frac{3}{2}, \frac{15}{13}, \frac{105}{76}, \frac{315}{263}, \frac{3465}{2578}, \frac{45045}{36979}, \ldots\right]$$

Type: Stream Fraction Integer

```
piConvergents := [4/p for p in ccf]
```

$$\left[4, \frac{8}{3}, \frac{52}{15}, \frac{304}{105}, \frac{1052}{315}, \frac{10312}{3465}, \frac{147916}{45045}, \ldots\right]$$

Type: Stream Fraction Integer

As you can see, the values are converging to $\pi = 3.14159265358979323846...$, but not very quickly.

```
piConvergents :: Stream Float
```

$$[4.0, 2.6666666666\ 666666667, 3.4666666666\ 666666667,$$
$$2.8952380952\ 380952381, 3.3396825396\ 825396825,$$
$$2.9760461760\ 461760462, 3.2837384837\ 384837385, \ldots]$$

Type: Stream Float

You need not restrict yourself to continued fractions of integers. Here is an expansion for a quotient of Gaussian integers.

```
continuedFraction((- 122 + 597*%i)/(4 - 4*%i))
```

$$-90 + 59\ i + \frac{1|}{|1 - 2\ i} + \frac{1|}{|-1 + 2\ i}$$

Type: ContinuedFraction Complex Integer

This is an expansion for a quotient of polynomials in one variable with rational number coefficients.

```
r : Fraction UnivariatePolynomial(x,Fraction Integer)
```

Type: Void

```
r := ((x - 1) * (x - 2)) / ((x-3) * (x-4))
```

$$\frac{x^2 - 3\ x + 2}{x^2 - 7\ x + 12}$$

Type: Fraction UnivariatePolynomial(x,Fraction Integer)

```
continuedFraction r
```

$$1 + \frac{1|}{\left|\frac{1}{4}\ x - \frac{9}{8}\right.} + \frac{1|}{\left|\frac{16}{3}\ x - \frac{40}{3}\right.}$$

```
Type: ContinuedFraction UnivariatePolynomial(x,Fraction Integer)
```

To conclude this section, we give you evidence that

$$z = 3 + \frac{1|}{|3} + \frac{1|}{|6} + \frac{1|}{|3} + \frac{1|}{|6} + \frac{1|}{|3} + \frac{1|}{|6} + \frac{1|}{|3} + \frac{1|}{|6} + \frac{1|}{|3} + \frac{1|}{|6} + \cdots$$

is the expansion of $\sqrt{11}$.

```
[i*i for i in convergents(z) :: Stream Float]
```

$$[9.0, 11.1111111111\ 11111111, 10.9944598337\ 9501385,$$

$$11.0002777777\ 77777778, 10.9999860763\ 98799786,$$

$$11.0000006979\ 29731039, 10.9999999650\ 15834446, \ldots]$$

```
Type: Stream Float
```

9.12 CycleIndicators

This section is based upon the paper J. H. Redfield, “The Theory of Group-Reduced Distributions”, American J. Math.,49 (1927) 433-455, and is an application of group theory to enumeration problems. It is a development of the work by P. A. MacMahon on the application of symmetric functions and Hammond operators to combinatorial theory.

The theory is based upon the power sum symmetric functions s_i which are the sum of the i-th powers of the variables. The cycle index of a permutation is an expression that specifies the sizes of the cycles of a permutation, and may be represented as a partition. A partition of a non-negative integer `n` is a collection of positive integers called its parts whose sum is `n`. For example, the partition $(3^2\ 2\ 1^2)$ will be used to represent $s_3^2 s_2 s_1^2$ and will indicate that the permutation has two cycles of length 3, one of length 2 and two of length 1. The cycle index of a permutation group is the sum of the cycle indices of its permutations divided by the number of permutations. The cycle indices of certain groups are provided.

We first load what we need from the library.

```
)load cycles evalcyc
```

```
library CYCLES has been loaded.
```

```
CycleIndicators is now explicitly exposed in frame G1077
```

The operation `complete` returns the cycle index of the symmetric group of order `n` for argument `n`. Alternatively, it is the n-th complete homogeneous symmetric function expressed in terms of power sum symmetric functions.

```
complete 1
```

$$(1)$$

Type: `SymmetricPolynomial Fraction Integer`

```
complete 2
```

$$\frac{1}{2}\ (2)+\frac{1}{2}\ \left(1^2\right)$$

Type: `SymmetricPolynomial Fraction Integer`

```
complete 3
```

$$\frac{1}{3}\ (3)+\frac{1}{2}\ (2\ \ 1)+\frac{1}{6}\ \left(1^3\right)$$

Type: `SymmetricPolynomial Fraction Integer`

```
complete 7
```

$$\tfrac{1}{7}\ (7)+\tfrac{1}{6}\ (6\ \ 1)+\tfrac{1}{10}\ (5\ \ 2)+\tfrac{1}{10}\ \left(5\ \ 1^2\right)+\tfrac{1}{12}\ (4\ \ 3)+\tfrac{1}{8}\ (4\ \ 2\ \ 1)+$$
$$\frac{1}{24}\ \left(4\ \ 1^3\right)+\frac{1}{18}\ \left(3^2\ 1\right)+\frac{1}{24}\ \left(3\ \ 2^2\right)+\frac{1}{12}\ \left(3\ \ 2\ \ 1^2\right)+\frac{1}{72}\ \left(3\ \ 1^4\right)+$$
$$\frac{1}{48}\ \left(2^3\ 1\right)+\frac{1}{48}\ \left(2^2\ 1^3\right)+\frac{1}{240}\ \left(2\ \ 1^5\right)+\frac{1}{5040}\ \left(1^7\right)$$

Type: `SymmetricPolynomial Fraction Integer`

The operation `elementary` computes the n-th elementary symmetric function for argument `n.`

```
elementary 7
```

$$\tfrac{1}{7}\ (7)-\tfrac{1}{6}\ (6\ \ 1)-\tfrac{1}{10}\ (5\ \ 2)+\tfrac{1}{10}\ \left(5\ \ 1^2\right)-\tfrac{1}{12}\ (4\ \ 3)+\tfrac{1}{8}\ (4\ \ 2\ \ 1)$$
$$-\frac{1}{24}\ \left(4\ \ 1^3\right)+\frac{1}{18}\ \left(3^2\ 1\right)+\frac{1}{24}\ \left(3\ \ 2^2\right)-\frac{1}{12}\ \left(3\ \ 2\ \ 1^2\right)+\frac{1}{72}\ \left(3\ \ 1^4\right)$$
$$-\frac{1}{48}\ \left(2^3\ 1\right)+\frac{1}{48}\ \left(2^2\ 1^3\right)-\frac{1}{240}\ \left(2\ \ 1^5\right)+\frac{1}{5040}\ \left(1^7\right)$$

Type: SymmetricPolynomial Fraction Integer

The operation `alternating` returns the cycle index of the alternating group having an even number of even parts in each cycle partition.

```
alternating 7
```

$$\frac{2}{7}\ (7) + \frac{1}{5}\ \left(5\ \ 1^2\right) + \frac{1}{4}\ (4\ \ 2\ \ 1) + \frac{1}{9}\ \left(3^2\ 1\right) + \frac{1}{12}\ \left(3\ \ 2^2\right) + \frac{1}{36}\ \left(3\ \ 1^4\right) + \frac{1}{24}\ \left(2^2\ 1^3\right) + \frac{1}{2520}\ \left(1^7\right)$$

Type: SymmetricPolynomial Fraction Integer

The operation `cyclic` returns the cycle index of the cyclic group.

```
cyclic 7
```

$$\frac{6}{7}\ (7) + \frac{1}{7}\ \left(1^7\right)$$

Type: SymmetricPolynomial Fraction Integer

The operation `dihedral` is the cycle index of the dihedral group.

```
dihedral 7
```

$$\frac{3}{7}\ (7) + \frac{1}{2}\ \left(2^3\ 1\right) + \frac{1}{14}\ \left(1^7\right)$$

Type: SymmetricPolynomial Fraction Integer

The operation `graphs` for argument `n` returns the cycle index of the group of permutations on the edges of the complete graph with `n` nodes induced by applying the symmetric group to the nodes.

```
graphs 5
```

$$\frac{1}{6}\ (6\ \ 3\ \ 1) + \frac{1}{5}\ \left(5^2\right) + \frac{1}{4}\ \left(4^2\ 2\right) + \frac{1}{6}\ \left(3^3\ 1\right) + \frac{1}{8}\ \left(2^4\ 1^2\right) + \frac{1}{12}\ \left(2^3\ 1^4\right) + \frac{1}{120}\ \left(1^{10}\right)$$

Type: SymmetricPolynomial Fraction Integer

The cycle index of a direct product of two groups is the product of the cycle indices of the groups. Redfield provided two operations on two cycle indices which will be called "cup" and "cap" here. The `cup` of two cycle indices is a kind of scalar product that combines monomials for permutations with the same cycles. The `cap` operation provides the sum of the coefficients of the result of the `cup` operation which will be an integer that enumerates what Redfield called group-reduced distributions.

We can, for example, represent `complete 2 * complete 2` as the set of objects `a a b b` and `complete 2 * complete 1 * complete 1` as `c c d e.`

This integer is the number of different sets of four pairs.

```
cap(complete 2**2, complete 2*complete 1**2)
```

$$4$$

Type: Fraction Integer

For example,

```
a a b b   a a b b   a a b b   a a b b
c c d e   c d c e   c e c d   d e c c
```

This integer is the number of different sets of four pairs no two pairs being equal.

```
cap(elementary 2**2, complete 2*complete 1**2)
```

$$2$$

Type: Fraction Integer

For example,

```
a a b b   a a b b
c d c e   c e c d
```

In this case the configurations enumerated are easily constructed, however the theory merely enumerates them providing little help in actually constructing them.

Here are the number of 6-pairs, first from `a a a b b c`, second from `d d e e f g`.

```
cap(complete 3*complete 2*complete 1,complete 2**2*complete 1**2)
```

$$24$$

Type: Fraction Integer

Here it is again, but with no equal pairs.

```
cap(elementary 3*elementary 2*elementary 1,complete 2**2*complete
1**2)
```

$$8$$

Type: Fraction Integer

```
cap(complete 3*complete 2*complete 1,elementary 2**2*elementary
1**2)
```

$$8$$

Type: Fraction Integer

The number of 6-triples, first from `a a a b b c`, second from `d d e e f g`, third from `h h i i j j`.

```
eval(cup(complete 3*complete 2*complete 1, cup(complete
2**2*complete 1**2,complete 2**3)))
```

$$1500$$

Type: Fraction Integer

The cycle index of vertices of a square is dihedral 4.

```
square:=dihedral 4
```

$$\frac{1}{4}\ (4) + \frac{3}{8}\ \left(2^2\right) + \frac{1}{4}\ \left(2\ \ 1^2\right) + \frac{1}{8}\ \left(1^4\right)$$

Type: SymmetricPolynomial Fraction Integer

The number of different squares with 2 red vertices and 2 blue vertices.

```
cap(complete 2**2,square)
```

$$2$$

Type: Fraction Integer

The number of necklaces with 3 red beads, 2 blue beads and 2 green beads.

```
cap(complete 3*complete 2**2,dihedral 7)
```

$$18$$

Type: Fraction Integer

The number of graphs with 5 nodes and 7 edges.

```
cap(graphs 5,complete 7*complete 3)
```

$$4$$

Type: Fraction Integer

The cycle index of rotations of vertices of a cube.

```
s(x) == powerSum(x)
```

Type: Void

```
cube:=(1/24)*(s 1**8+9*s 2**4 + 8*s 3**2*s 1**2+6*s 4**2)
```

```
   Compiling function s with type PositiveInteger ->
      SymmetricPolynomial Fraction Integer
```

$$\frac{1}{4}\ \left(4^2\right)+\frac{1}{3}\ \left(3^2\ 1^2\right)+\frac{3}{8}\ \left(2^4\right)+\frac{1}{24}\ \left(1^8\right)$$

Type: SymmetricPolynomial Fraction Integer

The number of cubes with 4 red vertices and 4 blue vertices.

```
cap(complete 4**2,cube)
```

$$7$$

Type: Fraction Integer

The number of labeled graphs with degree sequence `2 2 2 1 1` with no loops or multiple edges.

```
cap(complete 2**3*complete 1**2,wreath(elementary 4,elementary
2))
```

$$7$$

Type: Fraction Integer

Again, but with loops allowed but not multiple edges.

```
cap(complete 2**3*complete 1**2,wreath(elementary 4,complete 2))
```

$$17$$

Type: Fraction Integer

Again, but with multiple edges allowed, but not loops

```
cap(complete 2**3*complete 1**2,wreath(complete 4,elementary 2))
```

$$10$$

Type: Fraction Integer

Again, but with both multiple edges and loops allowed

```
cap(complete 2**3*complete 1**2,wreath(complete 4,complete 2))
```

$$23$$

Type: Fraction Integer

Having constructed a cycle index for a configuration we are at liberty to evaluate the s_i components any way we please. For example we can produce enumerating generating functions. This is done by providing a function `f` on an integer `i` to the value required of s_i, and then evaluating `eval(f, cycleindex)`.

```
x: ULS(FRAC INT,'x,0) := 'x
```

$$x$$

Type: UnivariateLaurentSeries(Fraction Integer,x,0)

```
ZeroOrOne: INT -> ULS(FRAC INT, 'x, 0)
```

Type: Void

```
Integers: INT -> ULS(FRAC INT, 'x, 0)
```

Type: Void

For the integers 0 and 1, or two colors.

```
ZeroOrOne n == 1+x**n
```

Type: Void

```
ZeroOrOne 5
```

$$1 + x^5$$

Type: UnivariateLaurentSeries(Fraction Integer,x,0)

For the integers `0, 1, 2, ...` we have this.

```
Integers n == 1/(1-x**n)
```

Type: Void

```
Integers 5
```

$$1+x^5+O\left(x^8\right)$$

Type: `UnivariateLaurentSeries(Fraction Integer,x,0)`

The coefficient of x^n is the number of graphs with 5 nodes and `n` edges.

```
eval(ZeroOrOne, graphs 5)
```

$$1+x+2\ x^2+4\ x^3+6\ x^4+6\ x^5+6\ x^6+4\ x^7+O\left(x^8\right)$$

Type: `UnivariateLaurentSeries(Fraction Integer,x,0)`

The coefficient of x^n is the number of necklaces with `n` red beads and `n-8` green beads.

```
eval(ZeroOrOne,dihedral 8)
```

$$1+x+4\ x^2+5\ x^3+8\ x^4+5\ x^5+4\ x^6+x^7+O\left(x^8\right)$$

Type: `UnivariateLaurentSeries(Fraction Integer,x,0)`

The coefficient of x^n is the number of partitions of `n` into 4 or fewer parts.

```
eval(Integers,complete 4)
```

$$1+x+2\ x^2+3\ x^3+5\ x^4+6\ x^5+9\ x^6+11\ x^7+O\left(x^8\right)$$

Type: `UnivariateLaurentSeries(Fraction Integer,x,0)`

The coefficient of x^n is the number of partitions of `n` into 4 boxes containing ordered distinct parts.

```
eval(Integers,elementary 4)
```

$$x^6+x^7+2\ x^8+3\ x^9+5\ x^{10}+6\ x^{11}+9\ x^{12}+11\ x^{13}+O\left(x^{14}\right)$$

Type: `UnivariateLaurentSeries(Fraction Integer,x,0)`

The coefficient of x^n is the number of different cubes with `n` red vertices and `8-n` green ones.

```
eval(ZeroOrOne,cube)
```

$$1+x+3\ x^2+3\ x^3+7\ x^4+3\ x^5+3\ x^6+x^7+O\left(x^8\right)$$

Type: `UnivariateLaurentSeries(Fraction Integer,x,0)`

The coefficient of x^n is the number of different cubes with integers on the vertices whose sum is `n`.

```
eval(Integers,cube)
```

$$1 + x + 4\ x^2 + 7\ x^3 + 21\ x^4 + 37\ x^5 + 85\ x^6 + 151\ x^7 + O\left(x^8\right)$$

Type: `UnivariateLaurentSeries(Fraction Integer,x,0)`

The coefficient of x^n is the number of graphs with 5 nodes and with integers on the edges whose sum is `n.` In other words, the enumeration is of multigraphs with 5 nodes and `n` edges.

```
eval(Integers,graphs 5)
```

$$1 + x + 3\ x^2 + 7\ x^3 + 17\ x^4 + 35\ x^5 + 76\ x^6 + 149\ x^7 + O\left(x^8\right)$$

Type: `UnivariateLaurentSeries(Fraction Integer,x,0)`

Graphs with 15 nodes enumerated with respect to number of edges.

```
eval(ZeroOrOne ,graphs 15)
```

$$1 + x + 2\ x^2 + 5\ x^3 + 11\ x^4 + 26\ x^5 + 68\ x^6 + 177\ x^7 + O\left(x^8\right)$$

Type: `UnivariateLaurentSeries(Fraction Integer,x,0)`

Necklaces with 7 green beads, 8 white beads, 5 yellow beads and 10 red beads.

```
cap(dihedral 30,complete 7*complete 8*complete 5*complete 10)
```

$$49958972383320$$

Type: `Fraction Integer`

The operation `SFunction` is the S-function or Schur function of a partition written as a descending list of integers expressed in terms of power sum symmetric functions.

In this case the argument partition represents a tableau shape. For example `3,2,2,1` represents a tableau with three boxes in the first row, two boxes in the second and third rows, and one box in the fourth row. `SFunction [3,2,2,1]` counts the number of different tableaux of shape `3, 2, 2, 1` filled with objects with an ascending order in the columns and a non-descending order in the rows.

```
sf3221:= SFunction [3,2,2,1]
```

$$\tfrac{1}{12}\ (6\ \ 2) - \tfrac{1}{12}\ \left(6\ \ 1^2\right) - \tfrac{1}{16}\ \left(4^2\right) + \tfrac{1}{12}\ (4\ \ 3\ \ 1) + \tfrac{1}{24}\ \left(4\ \ 1^4\right) - \tfrac{1}{36}\ \left(3^2\ 2\right) +$$
$$\frac{1}{36}\ \left(3^2\ 1^2\right) - \frac{1}{24}\ \left(3\ \ 2^2\ 1\right) - \frac{1}{36}\ \left(3\ \ 2\ \ 1^3\right) - \frac{1}{72}\ \left(3\ \ 1^5\right) - \frac{1}{192}\ \left(2^4\right) +$$
$$\frac{1}{48}\ \left(2^3\ 1^2\right) + \frac{1}{96}\ \left(2^2\ 1^4\right) - \frac{1}{144}\ \left(2\ \ 1^6\right) + \frac{1}{576}\ \left(1^8\right)$$

Type: SymmetricPolynomial Fraction Integer

This is the number filled with `a a b b c c d d`.

```
cap(sf3221,complete 2**4)
```

$$3$$

Type: Fraction Integer

The configurations enumerated above are:

```
a a b    a a c    a a d
b c      b b      b b
c d      c d      c c
d        d        d
```

This is the number of tableaux filled with `1..8`.

```
cap(sf3221, powerSum 1**8)
```

$$70$$

Type: Fraction Integer

The coefficient of x^n is the number of column strict reverse plane partitions of `n` of shape `3 2 2 1`.

```
eval(Integers, sf3221)
```

$$x^9 + 3\ x^{10} + 7\ x^{11} + 14\ x^{12} + 27\ x^{13} + 47\ x^{14} + O\left(x^{15}\right)$$

Type: UnivariateLaurentSeries(Fraction Integer,x,0)

The smallest is

```
0 0 0
1 1
2 2
3
```

9.13 DeRhamComplex

The domain constructor `DeRhamComplex` creates the class of differential forms of arbitrary degree over a coefficient ring. The De Rham complex constructor takes two arguments: a ring, `coefRing`, and a list of coordinate variables.

This is the ring of coefficients.

```
macro coefRing == Integer
```

Type: Void

These are the coordinate variables.

```
lv : List Symbol := [x,y,z]
```

$$[x, y, z]$$

Type: List Symbol

This is the De Rham complex of Euclidean three-space using coordinates `x`, `y` and `z`.

```
der := DERHAM(coefRing,lv)
```

$$DeRhamComplex(Integer, [x, y, z])$$

Type: Domain

This complex allows us to describe differential forms having expressions of integers as coefficients. These coefficients can involve any number of variables, for example, `f(x,t,r,y,u,z)`. As we've chosen to work with ordinary Euclidean three-space, expressions involving these forms are treated as functions of `x`, `y` and `z` with the additional arguments `t`, `r` and `u` regarded as symbolic constants.

Here are some examples of coefficients.

```
R := Expression coefRing
```

Expression Integer

Type: Domain

```
f : R := x**2*y*z-5*x**3*y**2*z**5
```

$$-5\ x^3\ y^2\ z^5 + x^2\ y\ z$$

Type: Expression Integer

```
g : R := z**2*y*cos(z)-7*sin(x**3*y**2)*z**2
```

$$-7\ z^2\ \sin\left(x^3\ y^2\right) + y\ z^2\ \cos\left(z\right)$$

Type: Expression Integer

```
h : R :=x*y*z-2*x**3*y*z**2
```

$$-2\ x^3\ y\ z^2 + x\ y\ z$$

Type: Expression Integer

We now define the multiplicative basis elements for the exterior algebra over R.

```
dx : der := generator(1)
```

$$dx$$

Type: DeRhamComplex(Integer,[x,y,z])

```
dy : der := generator(2)
```

$$dy$$

Type: DeRhamComplex(Integer,[x,y,z])

```
dz : der := generator(3)
```

$$dz$$

Type: DeRhamComplex(Integer,[x,y,z])

This is an alternative way to give the above assignments.

```
[dx,dy,dz] := [generator(i)$der for i in 1..3]
```

$$[dx, dy, dz]$$

Type: List DeRhamComplex(Integer,[x,y,z])

Now we define some one-forms.

```
alpha : der := f*dx + g*dy + h*dz
```

$$\left(-2\ x^3\ y\ z^2 + x\ y\ z\right)\ dz+$$
$$\left(-7\ z^2\ \sin\left(x^3\ y^2\right) + y\ z^2\ \cos\left(z\right)\right)\ dy+$$
$$\left(-5\ x^3\ y^2\ z^5 + x^2\ y\ z\right)\ dx$$

Type: DeRhamComplex(Integer,[x,y,z])

```
beta : der := cos(tan(x*y*z)+x*y*z)*dx + x*dy
```

$$x\ dy + \cos\left(\tan\left(x\ y\ z\right) + x\ y\ z\right)\ dx$$

Type: DeRhamComplex(Integer,[x,y,z])

A well-known theorem states that the composition of **exteriorDifferential** with itself is the zero map for continuous forms. Let's verify this theorem for `alpha`.

```
exteriorDifferential alpha;
```

Type: DeRhamComplex(Integer,[x,y,z])

We supressed the lengthy output of the last expression, but nevertheless, the composition is zero.

```
exteriorDifferential %
```

$$0$$

Type: DeRhamComplex(Integer,[x,y,z])

Now we check that **exteriorDifferential** is a "graded derivation" `D`, that is, `D` satisfies:

$$D(ab) = D(a)b + (-1)^{\deg(a)} aD(b)$$

```
gamma := alpha * beta
```

$$\begin{aligned}
&\left(2\ x^4\ y\ z^2 - x^2\ y\ z\right)\ dy\ dz+ \\
&\left(2\ x^3\ y\ z^2 - x\ y\ z\right)\ \cos\left(\tan\left(x\ y\ z\right) + x\ y\ z\right)\ dx\ dz+ \\
&\left(\left(7\ z^2\ \sin\left(x^3\ y^2\right) - y\ z^2\ \cos\left(z\right)\right)\ \cos\left(\tan\left(x\ y\ z\right) + x\ y\ z\right) - \right. \\
&\left. 5\ x^4\ y^2\ z^5 + x^3\ y\ z\right)\ dx\ dy
\end{aligned}$$

Type: DeRhamComplex(Integer,[x,y,z])

We try this for the one-forms `alpha` and `beta`.

```
exteriorDifferential(gamma) - (exteriorDifferential(alpha)*beta -
alpha * exteriorDifferential(beta))
```

$$0$$

Type: DeRhamComplex(Integer,[x,y,z])

Now we define some "basic operators" (see 9.45 on page 160).

```
a :  BOP := operator('a)
```

$$a$$

Type: BasicOperator

```
b : BOP := operator('b)
```

$$b$$

Type: BasicOperator

```
c : BOP := operator('c)
```

$$c$$

Type: BasicOperator

We also define some indeterminate one- and two-forms using these operators.

```
sigma := a(x,y,z) * dx + b(x,y,z) * dy + c(x,y,z) * dz
```

$$c\,(x,y,z)\ dz + b\,(x,y,z)\ dy + a\,(x,y,z)\ dx$$

Type: DeRhamComplex(Integer,[x,y,z])

```
theta := a(x,y,z) * dx * dy + b(x,y,z) * dx * dz + c(x,y,z) * dy
* dz
```

$$c\,(x,y,z)\ dy\ dz + b\,(x,y,z)\ dx\ dz + a\,(x,y,z)\ dx\ dy$$

Type: DeRhamComplex(Integer,[x,y,z])

This allows us to get formal definitions for the "gradient" ...

```
totalDifferential(a(x,y,z))$der
```

$$a_{,3}\,(x,y,z)\ dz + a_{,2}\,(x,y,z)\ dy + a_{,1}\,(x,y,z)\ dx$$

Type: DeRhamComplex(Integer,[x,y,z])

the "curl" ...

```
exteriorDifferential sigma
```

$$\begin{gathered}(c_{,2}\,(x,y,z) - b_{,3}\,(x,y,z))\ dy\ dz +\\ (c_{,1}\,(x,y,z) - a_{,3}\,(x,y,z))\ dx\ dz +\\ (b_{,1}\,(x,y,z) - a_{,2}\,(x,y,z))\ dx\ dy\end{gathered}$$

Type: DeRhamComplex(Integer,[x,y,z])

and the "divergence."

```
exteriorDifferential theta
```

$$(c_{,1}(x,y,z) - b_{,2}(x,y,z) + a_{,3}(x,y,z))\ dx\ dy\ dz$$

Type: DeRhamComplex(Integer,[x,y,z])

Note that the De Rham complex is an algebra with unity. This element `1` is the basis for elements for zero-forms, that is, functions in our space.

```
one : der := 1
```

$$1$$

Type: DeRhamComplex(Integer,[x,y,z])

To convert a function to a function lying in the De Rham complex, multiply the function by "one."

```
g1 : der := a([x,t,y,u,v,z,e]) * one
```

$$a(x,t,y,u,v,z,e)$$

Type: DeRhamComplex(Integer,[x,y,z])

A current limitation of AXIOM forces you to write functions with more than four arguments using square brackets in this way.

```
h1 : der := a([x,y,x,t,x,z,y,r,u,x]) * one
```

$$a(x,y,x,t,x,z,y,r,u,x)$$

Type: DeRhamComplex(Integer,[x,y,z])

Now note how the system keeps track of where your coordinate functions are located in expressions.

```
exteriorDifferential g1
```

$$a_{,6}(x,t,y,u,v,z,e)\ dz+$$

$$a_{,3}(x,t,y,u,v,z,e)\ dy+$$

$$a_{,1}(x,t,y,u,v,z,e)\ dx$$

Type: DeRhamComplex(Integer,[x,y,z])

```
exteriorDifferential h1
```

$$
\begin{array}{l}
a_{,6}\left(x, y, x, t, x, z, y, r, u, x\right)\ dz+ \\
(a_{,7}\left(x, y, x, t, x, z, y, r, u, x\right)+ \\
a_{,2}\left(x, y, x, t, x, z, y, r, u, x\right))\ \ dy+ \\
(a_{,10}\left(x, y, x, t, x, z, y, r, u, x\right)+ \\
a_{,5}\left(x, y, x, t, x, z, y, r, u, x\right)+ \\
a_{,3}\left(x, y, x, t, x, z, y, r, u, x\right)+ \\
a_{,1}\left(x, y, x, t, x, z, y, r, u, x\right))\ \ dx
\end{array}
$$

Type: DeRhamComplex(Integer,[x,y,z])

In this example of Euclidean three-space, the basis for the De Rham complex consists of the eight forms: `1`, `dx`, `dy`, `dz`, `dx*dy`, `dx*dz`, `dy*dz`, and `dx*dy*dz`.

```
coefficient(gamma, dx*dy)
```

$$
\begin{array}{l}
\left(7\ z^2\ \sin\left(x^3\ y^2\right) - y\ z^2\ \cos\left(z\right)\right)\ \cos\left(\tan\left(x\ y\ z\right) + x\ y\ z\right) \\
-5\ x^4\ y^2\ z^5 + x^3\ y\ z
\end{array}
$$

Type: Expression Integer

```
coefficient(gamma, one)
```

$$0$$

Type: Expression Integer

```
coefficient(g1,one)
```

$$a\left(x, t, y, u, v, z, e\right)$$

Type: Expression Integer

9.14 DecimalExpansion

All rationals have repeating decimal expansions. Operations to access the individual digits of a decimal expansion can be obtained by converting the value to `RadixExpansion(10)`. More examples of expansions are available in 9.3 on page 6, 9.29 on page 96, and 9.51 on page 184. Issue the system command `) show DecimalExpansion` to display the full list of operations defined by `DecimalExpansion`.

The operation `DecimalExpansion` is used to create this expansion of type `DecimaExpansion`.

```
r := decimal(22/7)
```

$$3.\overline{142857}$$

Type: DecimalExpansion

Arithmetic is exact.

```
r + decimal(6/7)
```

$$4$$

Type: DecimalExpansion

The period of the expansion can be short or long ...

```
[decimal(1/i) for i in 350..354]
```

$$\left[0.00\overline{285714}, 0.\overline{002849}, 0.0028\overline{409}, 0.\overline{00283286118980169971671388101983},\right.$$

$$\left.0.0\overline{0282485875706214689265536723163841807909604519774011299435}\right]$$

Type: List DecimalExpansion

or very long.

```
decimal(1/2049)
```

$$0.\overline{00048804294777940458760370912640312347486578818936066373840897999023 9}$$

$$\overline{1410444119082479258174719375305026842362127867252318204001952171791 11}$$

$$\overline{7618350414836505612493899463152757442654953635919960956564177647632 99}$$

$$\overline{1703269887750122010736944851146900927281600780868716447047340165934 60}$$

$$\overline{224499755978526110297706198145436798438262567105905319668130795 51}$$

Type: DecimalExpansion

These numbers are bona fide algebraic objects.

```
p := decimal(1/4)*x**2 + decimal(2/3)*x + decimal(4/9)
```

$$0.25\ x^2 + 0.\overline{6}\ x + 0.\overline{4}$$

Type: Polynomial DecimalExpansion

```
q := D(p, x)
```

$$0.5\ x + 0.\overline{6}$$

Type: Polynomial DecimalExpansion

```
g := gcd(p, q)
```

$$x + 1.\overline{3}$$

Type: Polynomial DecimalExpansion

9.15 DistributedMultivariatePolynomial

DistributedMultivariatePolynomial which is abbreviated as DMP and HomogeneousDistributedMultivariatePolynomial, which is abbreviated as HDMP, are very similar to MultivariatePolynomial except that they are represented and displayed in a non-recursive manner.

```
(d1,d2,d3) :  DMP([z,y,x],FRAC INT)
```

Type: Void

The constructor DMP orders its monomials lexicographically while HDMP orders them by total order refined by reverse lexicographic order.

```
d1 := -4*z + 4*y**2*x + 16*x**2 + 1
```

$$-4\ z + 4\ y^2\ x + 16\ x^2 + 1$$

Type: DistributedMultivariatePolynomial([z,y,x],Fraction Integer)

```
d2 := 2*z*y**2 + 4*x + 1
```

$$2\ z\ y^2 + 4\ x + 1$$

Type: DistributedMultivariatePolynomial([z,y,x],Fraction Integer)

```
d3 := 2*z*x**2 - 2*y**2 - x
```

$$2\ z\ x^2 - 2\ y^2 - x$$

Type: DistributedMultivariatePolynomial([z,y,x],Fraction Integer)

These constructors are mostly used in Gröbner basis calculations.

```
groebner [d1,d2,d3]
```

$$\left[z - \frac{1568}{2745}\ x^6 - \frac{1264}{305}\ x^5 + \frac{6}{305}\ x^4 + \frac{182}{549}\ x^3 - \frac{2047}{610}\ x^2 - \frac{103}{2745}\ x - \frac{2857}{10980}, \right.$$
$$y^2 + \frac{112}{2745}\ x^6 - \frac{84}{305}\ x^5 - \frac{1264}{305}\ x^4 - \frac{13}{549}\ x^3 + \frac{84}{305}\ x^2 + \frac{1772}{2745}\ x + \frac{2}{2745},$$
$$\left. x^7 + \frac{29}{4}\ x^6 - \frac{17}{16}\ x^4 - \frac{11}{8}\ x^3 + \frac{1}{32}\ x^2 + \frac{15}{16}\ x + \frac{1}{4}\right]$$

Type: List DistributedMultivariatePolynomial([z,y,x],Fraction Integer)

```
(n1,n2,n3) :  HDMP([z,y,x],FRAC INT)
```

Type: Void

```
(n1,n2,n3) := (d1,d2,d3)
```

$$2\ z\ x^2 - 2\ y^2 - x$$

Note that we get a different Gröbner basis when we use the HDMP polynomials, as expected.

```
groebner [n1,n2,n3]
```

$$\left[y^4 + 2\,x^3 - \frac{3}{2}\,x^2 + \frac{1}{2}\,z - \frac{1}{8},\right.$$

$$x^4 + \frac{29}{4}\,x^3 - \frac{1}{8}\,y^2 - \frac{7}{4}\,z\,x - \frac{9}{16}\,x - \frac{1}{4},$$

$$z\,y^2 + 2\,x + \frac{1}{2},$$

$$y^2\,x + 4\,x^2 - z + \frac{1}{4},$$

$$z\,x^2 - y^2 - \frac{1}{2}\,x,$$

$$\left.z^2 - 4\,y^2 + 2\,x^2 - \frac{1}{4}\,z - \frac{3}{2}\,x\right]$$

Type: List HomogeneousDistributedMultivariatePolynomial([z,y,x],Fraction Integer)

GeneralDistributedMultivariatePolynomial is somewhat more flexible in the sense that as well as accepting a list of variables to specify the variable ordering, it also takes a predicate on exponent vectors to specify the term ordering. With this polynomial type the user can experiment with the effect of using completely arbitrary term orderings. This flexibility is mostly important for algorithms such as Gröbner basis calculations which can be very sensitive to term ordering.

Issue the system command) show DistributedMultivariatePolynomial to display the full list of operations defined by DistributedMultivariatePolynomial.

9.16 EqTable

The EqTable domain provides tables where the keys are compared using **eq?**. Keys are considered equal only if they are the same instance of a structure. This is useful if the keys are themselves updatable structures. Otherwise, all operations are the same as for type Table. See 9.64 on page 215 for general information about tables. Issue the system command show EqTable to display the full list of operations defined by EqTable.

The operation **table** is here used to create a table where the keys are lists of integers.

```
e:  EqTable(List Integer, Integer) := table()
```

$$table()$$

Type: EqTable(List Integer,Integer)

These two lists are equal according to "=", but not according to **eq?**.

```
l1 := [1,2,3]
```

$$[1,2,3]$$

Type: List PositiveInteger

```
l2 := [1,2,3]
```

$$[1,2,3]$$

Type: List PositiveInteger

Because the two lists are not **eq?**, separate values can be stored under each.

```
e.l1 := 111
```

$$111$$

Type: PositiveInteger

```
e.l2 := 222
```

$$222$$

Type: PositiveInteger

```
e.l1
```

$$111$$

Type: PositiveInteger

9.17 Equation

The `Equation` domain provides equations as mathematical objects. These are used, for example, as the input to various **solve** operations.

Equations are created using the equals symbol, "`=`".

```
eq1 := 3*x + 4*y = 5
```

$$4\ y + 3\ x = 5$$

Type: Equation Polynomial Integer

```
eq2 := 2*x + 2*y = 3
```

$$2\ y + 2\ x = 3$$

Type: Equation Polynomial Integer

The left- and right-hand sides of an equation are accessible using the operations **lhs** and **rhs**.

```
lhs eq1
```

$$4\ y + 3\ x$$

Type: Polynomial Integer

```
rhs eq1
```

$$5$$

Type: Polynomial Integer

Arithmetic operations are supported and operate on both sides of the equation.

```
eq1 + eq2
```

$$6\ y + 5\ x = 8$$

Type: Equation Polynomial Integer

```
eq1 * eq2
```

$$8\ y^2 + 14\ x\ y + 6\ x^2 = 15$$

Type: Equation Polynomial Integer

```
2*eq2 - eq1
```

$$x = 1$$

Type: Equation Polynomial Integer

Equations may be created for any type so the arithmetic operations will be defined only when they make sense. For example, exponentiation is not defined for equations involving non-square matrices.

```
eq1**2
```

$$16\ y^2 + 24\ x\ y + 9\ x^2 = 25$$

Type: Equation Polynomial Integer

Note that an equals symbol is also used to *test* for equality of values in certain contexts. For example, `x+1` and `y` are unequal as polynomials.

```
if x+1 = y then "equal" else "unequal"
```

"unequal"

Type: String

```
eqpol := x+1 = y
```

$$x + 1 = y$$

Type: Equation Polynomial Integer

If an equation is used where a `Boolean` value is required, then it is evaluated using the equality test from the operand type.

```
if eqpol then "equal" else "unequal"
```

"unequal"

Type: String

If one wants a `Boolean` value rather than an equation, all one has to do is ask!

```
eqpol::Boolean
```

false

Type: Boolean

9.18 Exit

A function that does not return directly to its caller has `Exit` as its return type. The operation `error` is an example of one which does not return to its caller. Instead, it causes a return to top-level.

```
n := 0
```

$$0$$

```
Type: NonNegativeInteger
```

The function `gasp` is given return type `Exit` since it is guaranteed never to return a value to its caller.

```
gasp(): Exit ==
    free n
    n := n + 1
    error "Oh no!"
```

```
Function declaration gasp: ()-> Exit has been added to workspace.
```

```
Type: Void
```

The return type of `half` is determined by resolving the types of the two branches of the `if`.

```
half(k) ==
  if odd? k then gasp()
  else k quo 2
```

Because `gasp` has the return type `Exit`, the type of `if` in `half` is resolved to be `Integer`.

```
half 4
```

```
   Compiling function gasp with type () -> Exit
   Compiling function half with type PositiveInteger -> Integer
```

$$2$$

```
Type: PositiveInteger
```

```
half 3
```

```
Error signalled from user code in function gasp:
   Oh no!
```

```
n
```

$$1$$

Type: NonNegativeInteger

For functions which return no value at all, use `Void`.

9.19 Factored

`Factored` creates a domain whose objects are kept in factored form as long as possible. Thus certain operations like "`*`" (multiplication) and **gcd** are relatively easy to do. Others, such as addition, require somewhat more work, and the result may not be completely factored unless the argument domain `R` provides a **factor** operation. Each object consists of a unit and a list of factors, where each factor consists of a member of `R` (the *base*), an exponent, and a flag indicating what is known about the base. A flag may be one of "`nil`", "`sqfr`", "`irred`" or "`prime`", which mean that nothing is known about the base, it is square-free, it is irreducible, or it is prime, respectively. The current restriction to factored objects of integral domains allows simplification to be performed without worrying about multiplication order.

9.19.1 Decomposing Factored Objects

In this section we will work with a factored integer.

```
g := factor(4312)
```

$$2^3\ 7^2\ 11$$

Type: Factored Integer

Let's begin by decomposing `g` into pieces. The only possible units for integers are `1` and `-1`.

```
unit(g)
```

$$1$$

Type: PositiveInteger

There are three factors.

```
numberOfFactors(g)
```

$$3$$

Type: PositiveInteger

We can make a list of the bases, ...

```
[nthFactor(g,i) for i in 1..numberOfFactors(g)]
```

$$[2, 7, 11]$$

Type: List Integer

and the exponents, ...

```
[nthExponent(g,i) for i in 1..numberOfFactors(g)]
```

$$[3, 2, 1]$$

Type: List Integer

and the flags. You can see that all the bases (factors) are prime.

```
[nthFlag(g,i) for i in 1..numberOfFactors(g)]
```

$$[\texttt{"prime"}, \texttt{"prime"}, \texttt{"prime"}]$$

Type: List Union("nil","sqfr","irred","prime")

A useful operation for pulling apart a factored object into a list of records of the components is **factorList**.

```
factorList(g)
```

$$[[flg = \texttt{"prime"}, fctr = 2, xpnt = 3],$$

$$[flg = \texttt{"prime"}, fctr = 7, xpnt = 2],$$

$$[flg = \texttt{"prime"}, fctr = 11, xpnt = 1]]$$

Type: List Record(flg: Union("nil","sqfr","irred","prime"),
fctr: Integer,xpnt: Integer)

If you don't care about the flags, use **factors**.

```
factors(g)
```

$$[[factor = 2, exponent = 3],$$

$$[factor = 7, exponent = 2],$$

$$[factor = 11, exponent = 1]]$$

Type: List Record(factor: Integer,exponent: Integer)

Neither of these operations returns the unit.

```
first(%).factor
```

$$2$$

Type: PositiveInteger

9.19.2 Expanding Factored Objects

Recall that we are working with this factored integer.

```
g := factor(4312)
```

$$2^3\ 7^2\ 11$$

Type: Factored Integer

To multiply out the factors with their multiplicities, use **expand**.

```
expand(g)
```

$$4312$$

Type: PositiveInteger

If you would like, say, the distinct factors multiplied together but with multiplicity one, you could do it this way.

```
reduce(*,[t.factor for t in factors(g)])
```

$$154$$

Type: PositiveInteger

9.19.3 Arithmetic with Factored Objects

We're still working with this factored integer.

```
g := factor(4312)
```

$$2^3\ 7^2\ 11$$

Type: Factored Integer

We'll also define this factored integer.

```
f := factor(246960)
```

$$2^4\ 3^2\ 5\ 7^3$$

Type: Factored Integer

Operations involving multiplication and division are particularly easy with factored objects.

```
f * g
```

$$2^7\ 3^2\ 5\ 7^5\ 11$$

Type: Factored Integer

```
f**500
```

$$2^{2000}\ 3^{1000}\ 5^{500}\ 7^{1500}$$

Type: Factored Integer

```
gcd(f,g)
```

$$2^3\ 7^2$$

Type: Factored Integer

```
lcm(f,g)
```

$$2^4\ 3^2\ 5\ 7^3\ 11$$

Type: Factored Integer

If we use addition and subtraction things can slow down because we may need to compute greatest common divisors.

```
f + g
```

$$2^3\ 7^2\ 641$$

Type: Factored Integer

```
f - g
```

$$2^3\ 7^2\ 619$$

Type: Factored Integer

Test for equality with `0` and `1` by using **zero?** and **one?**, respectively.

```
zero?(factor(0))
```

true

Type: Boolean

```
zero?(g)
```

false

Type: Boolean

```
one?(factor(1))
```

true

Type: Boolean

```
one?(f)
```

false

Type: Boolean

Another way to get the zero and one factored objects is to use package calling.

```
0$Factored(Integer)
```

$$0$$

Type: Factored Integer

```
1$Factored(Integer)
```

$$1$$

Type: Factored Integer

9.19.4 Creating New Factored Objects

The **map** operation is used to iterate across the unit and bases of a factored object.

The following four operations take a base and an exponent and create a factored object. They differ in handling the flag component.

```
nilFactor(24,2)
```

$$24^2$$

Type: Factored Integer

This factor has no associated information.

```
nthFlag(%,1)
```

"nil"

Type: Union("nil",...)

This factor is asserted to be square-free.

```
sqfrFactor(12,2)
```

$$12^2$$

Type: Factored Integer

This factor is asserted to be irreducible.

```
irreducibleFactor(13,10)
```

$$13^{10}$$

Type: Factored Integer

This factor is asserted to be prime.

```
primeFactor(11,5)
```

$$11^5$$

Type: Factored Integer

A partial inverse to **factorList** is **makeFR**.

```
h := factor(-720)
```

$$-2^4\ 3^2\ 5$$

Type: Factored Integer

The first argument is the unit and the second is a list of records as returned by **factorList**.

```
h - makeFR(unit(h),factorList(h))
```

$$0$$

Type: Factored Integer

9.19.5 Factored Objects with Variables

Some of the operations available for polynomials are also available for factored polynomials.

```
p := (4*x*x-12*x+9)*y*y + (4*x*x-12*x+9)*y + 28*x*x - 84*x + 63
```

$$\left(4\ x^2 - 12\ x + 9\right)\ y^2 + \left(4\ x^2 - 12\ x + 9\right)\ y + 28\ x^2 - 84\ x + 63$$

Type: Polynomial Integer

```
fp := factor(p)
```

$$(2\ x - 3)^2\ \left(y^2 + y + 7\right)$$

Type: Factored Polynomial Integer

You can differentiate with respect to a variable.

```
D(p,x)
```

$$(8\ x - 12)\ y^2 + (8\ x - 12)\ y + 56\ x - 84$$

Type: Polynomial Integer

```
D(fp,x)
```

$$4\ (2\ x - 3)\ \left(y^2 + y + 7\right)$$

Type: Factored Polynomial Integer

```
numberOfFactors(%)
```

$$3$$

Type: PositiveInteger

9.20 FactoredFunctions2

The `FactoredFunctions2` package implements one operation, **map**, for applying an operation to every base in a factored object and to the unit.

```
double(x) == x + x
```

Type: Void

```
f := factor(720)
```

$$2^4\ 3^2\ 5$$

Type: Factored Integer

Actually, the **map** operation used in this example comes from `Factored` itself, since `double` takes an integer argument and returns an integer result.

```
map(double,f)
```

$$2\ 4^4\ 6^2\ 10$$

Type: Factored Integer

If we want to use an operation that returns an object that has a type different from the operation's argument, the **map** in `Factored` cannot be used and we use the one in `FactoredFunctions2`.

```
makePoly(b) == x + b
```

Type: Void

In fact, the "2" in the name of the package means that we might be using factored objects of two different types.

```
g := map(makePoly,f)
```

$$(x+1)\ (x+2)^4\ (x+3)^2\ (x+5)$$

Type: Factored Polynomial Integer

It is important to note that both versions of **map** destroy any information known about the bases (the fact that they are prime, for instance).

The flags for each base are set to "nil" in the object returned by **map**.

```
nthFlag(g,1)
```

"nil"

Type: Union("nil",...)

9.21 File

The `File(S)` domain provides a basic interface to read and write values of type `S` in files.

Before working with a file, it must be made accessible to AXIOM with the **open** operation.

```
ifile:File List Integer:=open("/tmp/jazz1","output")
```

"/tmp/jazz1"

Type: File List Integer

The **open** function arguments are a `FileName` and a `String` specifying the mode. If a full pathname is not specified, the current default directory is assumed. The mode must be one of "`input`" or "`output`". If it is not specified, "`input`" is assumed. Once the file has been opened, you can read or write data.

The operations **read** and **write** are provided.

```
write!(ifile, [-1,2,3])
```

$$[-1, 2, 3]$$

Type: List Integer

```
write!(ifile, [10,-10,0,111])
```

$$[10, -10, 0, 111]$$

Type: List Integer

```
write!(ifile, [7])
```

$$[7]$$

Type: List Integer

You can change from writing to reading (or vice versa) by reopening a file.

```
reopen!(ifile, "input")
```

"/tmp/jazz1"

Type: File List Integer

```
read! ifile
```

$$[-1, 2, 3]$$

Type: List Integer

```
read! ifile
```

$$[10, -10, 0, 111]$$

Type: List Integer

The **read** operation can cause an error if one tries to read more data than is in the file. To guard against this possibility the **readIfCan** operation should be used.

```
readIfCan! ifile
```

$$[7]$$

Type: Union(List Integer,...)

```
readIfCan! ifile
```

"failed"

Type: Union("failed",...)

You can find the current mode of the file, and the file's name.

```
iomode ifile
```

"input"

Type: String

```
name ifile
```

"/tmp/jazz1"

Type: FileName

When you are finished with a file, you should close it.

```
close! ifile
```

"/tmp/jazz1"

Type: File List Integer

```
)system rm /tmp/jazz1
```

A limitation of the underlying LISP system is that not all values can be represented in a file. In particular, delayed values containing compiled functions cannot be saved.

For more information on related topics, see 9.65 on page 219, 9.33 on page 112, 9.34 on page 116, and 9.22 on page 76. Issue the system command `)show File` to the display the full list of operations defined by `File`.

9.22 FileName

The `FileName` domain provides an interface to the computer's file system. Functions are provided to manipulate file names and to test properties of files.

The simplest way to use file names in the AXIOM interpreter is to rely on conversion to and from strings. The syntax of these strings depends on the operating system.

```
fn: FileName
```

Type: Void

On AIX, this is a proper file syntax:

```
fn := "/spad/src/input/fname.input"
```

"/spad/src/input/fname.input"

Type: FileName

Although it is very convenient to be able to use string notation for file names in the interpreter, it is desirable to have a portable way of creating and manipulating file names from within programs.

A measure of portability is obtained by considering a file name to consist of three parts: the *directory*, the *name*, and the *extension*.

```
directory fn
```

"/spad/src/input"

Type: String

```
name fn
```

"fname"

Type: String

```
extension fn
```

"input"

Type: String

The meaning of these three parts depends on the operating system. For example, on CMS the file "`SPADPROF INPUT M`" would have directory "`M`", name "`SPADPROF`" and extension "`INPUT`".

It is possible to create a filename from its parts.

```
fn := filename("/u/smwatt/work", "fname", "input")
```

"/u/smwatt/work/fname.input"

Type: FileName

When writing programs, it is helpful to refer to directories via variables.

```
objdir := "/tmp"
```

"/tmp"

Type: String

```
fn := filename(objdir, "table", "spad")
```

"/tmp/table.spad"

Type: FileName

If the directory or the extension is given as an empty string, then a default is used. On AIX, the defaults are the current directory and no extension.

```
fn := filename("", "letter", "")
```

"letter"

Type: FileName

Three tests provide information about names in the file system.
The **exists?** operation tests whether the named file exists.

```
exists? "/etc/passwd"
```

true

Type: Boolean

The operation **readable?** tells whether the named file can be read. If the file does not exist, then it cannot be read.

```
readable? "/etc/passwd"
```

true

Type: Boolean

```
readable? "/etc/security/passwd"
```

```
                              false
                                                   Type: Boolean
```

```
readable? "/ect/passwd"
```

```
                              false
                                                   Type: Boolean
```

Likewise, the operation **writable?** tells whether the named file can be written. If the file does not exist, the test is determined by the properties of the directory.

```
writable? "/etc/passwd"
```

```
                              false
                                                   Type: Boolean
```

```
writable? "/dev/null"
```

```
                              true
                                                   Type: Boolean
```

```
writable? "/etc/DoesNotExist"
```

```
                              false
                                                   Type: Boolean
```

```
writable? "/tmp/DoesNotExist"
```

```
                              true
                                                   Type: Boolean
```

The **new** operation constructs the name of a new writable file. The argument sequence is the same as for **filename**, except that the name part is actually a prefix for a constructed unique name.

The resulting file is in the specified directory with the given extension, and the same defaults are used.

```
fn := new(objdir, "xxx", "yy")
```

```
                         "/tmp/xxx00007.yy"
                                                  Type: FileName
```

9.23 FlexibleArray

The `FlexibleArray` domain constructor creates one-dimensional arrays of elements of the same type. Flexible arrays are an attempt to provide a data type that has the best features of both one-dimensional arrays (fast, random access to elements) and lists (flexibility). They are implemented by a fixed block of storage. When necessary for expansion, a new, larger block of storage is allocated and the elements from the old storage area are copied into the new block.

Flexible arrays have available most of the operations provided by `OneDimensionalArray` (see 9.44 on page 158 and 9.69 on page 233). Since flexible arrays are also of category `ExtensibleLinearAggregate`, they have operations `concat!`, `delete!`, `insert!`, `merge!`, `remove!`, `removeDuplicates!`, and `select!`. In addition, the operations `physicalLength` and `physicalLength!` provide user-control over expansion and contraction.

A convenient way to create a flexible array is to apply the operation `flexibleArray` to a list of values.

```
flexibleArray [i for i in 1..6]
```

$$[1, 2, 3, 4, 5, 6]$$

Type: `FlexibleArray PositiveInteger`

Create a flexible array of six zeroes.

```
f : FARRAY INT := new(6,0)
```

$$[0, 0, 0, 0, 0, 0]$$

Type: `FlexibleArray Integer`

For $i = 1 \ldots 6$ set the i-th element to i. Display `f`.

```
for i in 1..6 repeat f.i := i; f
```

$$[1, 2, 3, 4, 5, 6]$$

Type: `FlexibleArray Integer`

Initially, the physical length is the same as the number of elements.

```
physicalLength f
```

$$6$$

Type: `PositiveInteger`

Add an element to the end of `f`.

```
concat!(f,11)
```

$$[1, 2, 3, 4, 5, 6, 11]$$

Type: FlexibleArray Integer

See that its physical length has grown.

```
physicalLength f
```

$$10$$

Type: PositiveInteger

Make `f` grow to have room for `15` elements.

```
physicalLength!(f,15)
```

$$[1, 2, 3, 4, 5, 6, 11]$$

Type: FlexibleArray Integer

Concatenate the elements of `f` to itself. The physical length allows room for three more values at the end.

```
concat!(f,f)
```

$$[1, 2, 3, 4, 5, 6, 11, 1, 2, 3, 4, 5, 6, 11]$$

Type: FlexibleArray Integer

Use `insert!` to add an element to the front of a flexible array.

```
insert!(22,f,1)
```

$$[22, 1, 2, 3, 4, 5, 6, 11, 1, 2, 3, 4, 5, 6, 11]$$

Type: FlexibleArray Integer

Create a second flexible array from `f` consisting of the elements from index 10 forward.

```
g := f(10..)
```

$$[2, 3, 4, 5, 6, 11]$$

Type: FlexibleArray Integer

Insert this array at the front of `f`.

```
insert!(g,f,1)
```

$$[2,3,4,5,6,11,22,1,2,3,4,5,6,11,1,2,3,4,5,6,11]$$

Type: FlexibleArray Integer

Merge the flexible array `f` into `g` after sorting each in place.

```
merge!(sort!  f, sort!  g)
```

$$[1,1,2,2,2,2,3,3,3,3,4,4,4,4,5,5,5,5,6,6,6,6,11,11,11,11,22]$$

Type: FlexibleArray Integer

Remove duplicates in place.

```
removeDuplicates!  f
```

$$[1,2,3,4,5,6,11,22]$$

Type: FlexibleArray Integer

Remove all odd integers.

```
select!(i +-> even?  i,f)
```

$$[2,4,6,22]$$

Type: FlexibleArray Integer

All these operations have shrunk the physical length of `f`.

```
physicalLength f
```

$$8$$

Type: PositiveInteger

To force AXIOM not to shrink flexible arrays call the `shrinkable` operation with the argument `false`. You must package call this operation. The previous value is returned.

```
shrinkable(false)$FlexibleArray(Integer)
```

`true`

Type: Boolean

9.24 Float

AXIOM provides two kinds of floating point numbers. The domain `Float` (abbreviation `FLOAT`) implements a model of arbitrary precision floating point numbers. The domain `DoubleFloat` (abbreviation `DFLOAT`) is intended to make available hardware floating point arithmetic in AXIOM. The actual model of floating point that `DoubleFloat` provides is system-dependent. For example, on the IBM system 370 AXIOM uses IBM double precision which has fourteen hexadecimal digits of precision or roughly sixteen decimal digits. Arbitrary precision floats allow the user to specify the precision at which arithmetic operations are computed. Although this is an attractive facility, it comes at a cost. Arbitrary-precision floating-point arithmetic typically takes twenty to two hundred times more time than hardware floating point.

9.24.1 Introduction to Float

Scientific notation is supported for input and output of floating point numbers. A floating point number is written as a string of digits containing a decimal point optionally followed by the letter "`E`", and then the exponent.

We begin by doing some calculations using arbitrary precision floats. The default precision is twenty decimal digits.

```
1.234
```

$$1.234$$

Type: `Float`

A decimal base for the exponent is assumed, so the number `1.234E2` denotes $1.234 \cdot 10^2$.

```
1.234E2
```

$$123.4$$

Type: `Float`

The normal arithmetic operations are available for floating point numbers.

```
sqrt(1.2 + 2.3 / 3.4 ** 4.5)
```

$$1.0996972790\ 671286226$$

Type: `Float`

9.24.2 Conversion Functions

You can use conversion to go back and forth between `Integer`, `Fraction Integer` and `Float`, as appropriate.

```
i := 3 :: Float
```

$$3.0$$

Type: Float

```
i :: Integer
```

$$3$$

Type: Integer

```
i :: Fraction Integer
```

$$3$$

Type: Fraction Integer

Since you are explicitly asking for a conversion, you must take responsibility for any loss of exactness.

```
r := 3/7 :: Float
```

$$0.4285714285\ 7142857143$$

Type: Float

```
r :: Fraction Integer
```

$$\frac{3}{7}$$

Type: Fraction Integer

This conversion cannot be performed: use **truncate** or **round** if that is what you intend.

```
r :: Integer
```

```
Cannot convert from type Float to Integer for value
0.4285714285 7142857143
```

The operations **truncate** and **round** truncate . . .

```
truncate 3.6
```

$$3.0$$

Type: `Float`

and round to the nearest integral `Float` respectively.

```
round 3.6
```

$$4.0$$

Type: `Float`

```
truncate(-3.6)
```

$$-3.0$$

Type: `Float`

```
round(-3.6)
```

$$-4.0$$

Type: `Float`

The operation **fractionPart** computes the fractional part of `x`, that is, `x - truncate x`.

```
fractionPart 3.6
```

$$0.6$$

Type: `Float`

The operation **digits** allows the user to set the precision. It returns the previous value it was using.

```
digits 40
```

$$20$$

Type: `PositiveInteger`

```
sqrt 0.2
```

$$0.4472135954\ 9995793928\ 1834733746\ 2552470881$$

Type: Float

```
pi()$Float
```

3.1415926535 8979323846 2643383279 502884197

Type: Float

The precision is only limited by the computer memory available. Calculations at 500 or more digits of precision are not difficult.

```
digits 500
```

40

Type: PositiveInteger

```
pi()$Float
```

3.1415926535 8979323846 2643383279 5028841971 6939937510 5820974944
5923078164 0628620899 8628034825 3421170679 8214808651 3282306647
0938446095 5058223172 5359408128 4811174502 8410270193 8521105559
6446229489 5493038196 4428810975 6659334461 2847564823 3786783165
2712019091 4564856692 3460348610 4543266482 1339360726 0249141273
7245870066 0631558817 4881520920 9628292540 9171536436 7892590360
0113305305 4882046652 1384146951 9415116094 3305727036 5759591953
0921861173 8193261179 3105118548 0744623799 6274956735 1885752724
8912279381 830119491

Type: Float

Reset **digits** to its default value.

```
digits 20
```

500

Type: PositiveInteger

Numbers of type `Float` are represented as a record of two integers, namely, the mantissa and the exponent where the base of the exponent is binary. That is, the floating point number `(m,e)` represents the number $m \cdot 2^e$. A consequence of using a binary base is that decimal numbers can not, in general, be represented exactly.

9.24.3 Output Functions

A number of operations exist for specifying how numbers of type `Float` are to be displayed. By default, spaces are inserted every ten digits in the output for readability.[3]

Output spacing can be modified with the **outputSpacing** operation. This inserts no spaces and then displays the value of `x`.

```
outputSpacing 0; x := sqrt 0.2
```

$$0.44721359549995793928$$

Type: Float

Issue this to have the spaces inserted every `5` digits.

```
outputSpacing 5; x
```

$$0.44721\ 35954\ 99957\ 93928$$

Type: Float

By default, the system displays floats in either fixed format or scientific format, depending on the magnitude of the number.

```
y := x/10**10
```

$$0.44721\ 35954\ 99957\ 93928\ \mathrm{E}\ -10$$

Type: Float

A particular format may be requested with the operations **outputFloating** and **outputFixed**.

```
outputFloating(); x
```

$$0.44721\ 35954\ 99957\ 93928\ \mathrm{E}\ 0$$

Type: Float

```
outputFixed(); y
```

$$0.00000\ 00000\ 44721\ 35954\ 99957\ 93928$$

Type: Float

Additionally, you can ask for `n` digits to be displayed after the decimal point.

[3]Note that you cannot include spaces in the input form of a floating point number, though you can use underscores.

```
outputFloating 2; y
```

$$0.45\ \mathrm{E}\ -10$$

Type: Float

```
outputFixed 2; x
```

$$0.45$$

Type: Float

This resets the output printing to the default behavior.

```
outputGeneral()
```

Type: Void

9.24.4 An Example: Determinant of a Hilbert Matrix

Consider the problem of computing the determinant of a `10` by `10` Hilbert matrix. The (i, j)-th entry of a Hilbert matrix is given by `1/(i+j+1)`.

First do the computation using rational numbers to obtain the exact result.

```
a:  Matrix Fraction Integer := matrix [ [1/(i+j+1) for j in 0..9]
for i in 0..9]
```

$$\begin{bmatrix}
1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} \\
\frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} \\
\frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} \\
\frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} \\
\frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} \\
\frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} \\
\frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} \\
\frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} \\
\frac{1}{9} & \frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} \\
\frac{1}{10} & \frac{1}{11} & \frac{1}{12} & \frac{1}{13} & \frac{1}{14} & \frac{1}{15} & \frac{1}{16} & \frac{1}{17} & \frac{1}{18} & \frac{1}{19}
\end{bmatrix}$$

Type: Matrix Fraction Integer

This version of **determinant** uses Gaussian elimination.

```
d:= determinant a
```

$$\frac{1}{46206893947914691316295628839036278726983680000000000}$$

Type: Fraction Integer

```
d :: Float
```

$$0.21641\ 79226\ 43149\ 18691\ \mathrm{E}\ -52$$

Type: Float

Now use hardware floats. Note that a semicolon (;) is used to prevent the display of the matrix.

```
b: Matrix DoubleFloat := matrix [ [1/(i+j+1$DoubleFloat) for j
in 0..9] for i in 0..9];
```

Type: Matrix DoubleFloat

The result given by hardware floats is correct only to four significant digits of precision. In the jargon of numerical analysis, the Hilbert matrix is said to be "ill-conditioned."

```
determinant b
```

$$2.1643677945721411e-53$$

Type: DoubleFloat

Now repeat the computation at a higher precision using `Float`.

```
digits 40
```

$$20$$

Type: PositiveInteger

```
c: Matrix Float := matrix [ [1/(i+j+1$Float) for j in 0..9] for
i in 0..9];
```

Type: Matrix Float

```
determinant c
```

$$0.21641\ 79226\ 43149\ 18690\ 60594\ 98362\ 26174\ 36159\ \mathrm{E}\ -52$$

Type: Float

Reset **digits** to its default value.

```
digits 20
```

$$40$$

Type: PositiveInteger

9.25 Fraction

The `Fraction` domain implements quotients. The elements must belong to a domain of category `IntegralDomain`: multiplication must be commutative and the product of two non-zero elements must not be zero. This allows you to make fractions of most things you would think of, but don't expect to create a fraction of two matrices! The abbreviation for `Fraction` is `FRAC`.

Use "`/`" to create a fraction.

```
a := 11/12
```

$$\frac{11}{12}$$

Type: `Fraction Integer`

```
b := 23/24
```

$$\frac{23}{24}$$

Type: `Fraction Integer`

The standard arithmetic operations are available.

```
3 - a*b**2 + a + b/a
```

$$\frac{313271}{76032}$$

Type: `Fraction Integer`

Extract the numerator and denominator by using **numer** and **denom**, respectively.

```
numer(a)
```

$$11$$

Type: `PositiveInteger`

```
denom(b)
```

$$24$$

Type: `PositiveInteger`

Operations like **max**, **min**, **negative?**, **positive?** and **zero?** are all available if they are provided for the numerators and denominators. See 9.30 on page 97 for examples.

Don't expect a useful answer from **factor**, **gcd** or **lcm** if you apply them to fractions.

```
r := (x**2 + 2*x + 1)/(x**2 - 2*x + 1)
```

$$\frac{x^2 + 2\ x + 1}{x^2 - 2\ x + 1}$$

Type: Fraction Polynomial Integer

Since all non-zero fractions are invertible, these operations have trivial definitions.

```
factor(r)
```

$$\frac{x^2 + 2\ x + 1}{x^2 - 2\ x + 1}$$

Type: Factored Fraction Polynomial Integer

Use **map** to apply **factor** to the numerator and denominator, which is probably what you mean.

```
map(factor,r)
```

$$\frac{(x+1)^2}{(x-1)^2}$$

Type: Fraction Factored Polynomial Integer

Other forms of fractions are available. Use `continuedFraction` to create a continued fraction.

```
continuedFraction(7/12)
```

$$\frac{1|}{|1} + \frac{1|}{|1} + \frac{1|}{|2} + \frac{1|}{|2}$$

Type: ContinuedFraction Integer

Use `partialFraction` to create a partial fraction. See 9.11 on page 37 and 9.47 on page 170 for additional information and examples.

```
partialFraction(7,12)
```

$$1 - \frac{3}{2^2} + \frac{1}{3}$$

Type: PartialFraction Integer

Use conversion to create alternative views of fractions with objects moved in and out of the numerator and denominator.

```
g := 2/3 + 4/5*%i
```

$$\frac{2}{3} + \frac{4}{5}\ \%i$$

Type: Complex Fraction Integer

```
g :: FRAC COMPLEX INT
```

$$\frac{10 + 12\ \%i}{15}$$

Type: Fraction Complex Integer

9.26 GeneralSparseTable

Sometimes when working with tables there is a natural value to use as the entry in all but a few cases. The `GeneralSparseTable` constructor can be used to provide any table type with a default value for entries. See 9.64 on page 215 for general information about tables. Issue the system command `) show GeneralSparseTable` to display the full list of operations defined by `GeneralSparseTable`.

Suppose we launched a fund-raising campaign to raise fifty thousand dollars. To record the contributions, we want a table with strings as keys (for the names) and integer entries (for the amount). In a data base of cash contributions, unless someone has been explicitly entered, it is reasonable to assume they have made a zero dollar contribution.

This creates a keyed access file with default entry `0`.

```
patrons: GeneralSparseTable(String, Integer,
KeyedAccessFile(Integer), 0) := table() ;


kaf00056.sdata"
```

Type: GeneralSparseTable(String,Integer,KeyedAccessFile Integer,0)

Now `patrons` can be used just as any other table. Here we record two gifts.

```
patrons."Smith" := 10500
```

$$10500$$

Type: PositiveInteger

```
patrons."Jones" := 22000
```

$$22000$$

Type: PositiveInteger

Now let us look up the size of the contributions from Jones and Stingy.

```
patrons."Jones"
```

$$22000$$

Type: PositiveInteger

```
patrons."Stingy"
```

$$0$$

Type: NonNegativeInteger

Have we met our seventy thousand dollar goal?

```
reduce(+, entries patrons)
```

$$32500$$

Type: PositiveInteger

So the project is cancelled and we can delete the data base:

```
)system rm -r kaf*.sdata
```

9.27 GroebnerFactorizationPackage

Solving systems of polynomial equations with the Gröbner basis algorithm can often be very time consuming because, in general, the algorithm has exponential run-time. These systems, which often come from concrete applications, frequently have symmetries which are not taken advantage of by the algorithm. However, it often happens in this case that the polynomials which occur during the Gröbner calculations are reducible. Since AXIOM has an excellent polynomial factorization algorithm, it is very natural to combine the Gröbner and factorization algorithms.

`GroebnerFactorizationPackage` exports the **groebnerFactorize** operation which implements a modified Gröbner basis algorithm. In this algorithm, each polynomial that is to be put into the partial list of the basis is first factored. The remaining calculation is split into as many parts as there are irreducible factors. Call these factors $p_1, \ldots, p_n$. In the branches corresponding to $p_2, \ldots, p_n$, the factor p_1 can be divided out, and so on. This package also contains operations that allow you to specify the polynomials that are not zero on the common roots of the final Gröbner basis.

Here is an example from chemistry. In a theoretical model of the cyclohexan C_6H_{12}, the six carbon atoms each sit in the center of gravity of a tetrahedron that has two hydrogen atoms and two carbon atoms at its corners. We first normalize and set the length of each edge to 1. Hence, the distances of one fixed carbon atom to each of its immediate neighbours is 1. We will denote the distances to the other three carbon atoms by x, y and z.

A. Dress developed a theory to decide whether a set of points and distances between them can be realized in an n-dimensional space. Here, of course, we have $n = 3$.

```
mfzn :  SQMATRIX(6,DMP([x,y,z],Fraction INT)) := [ [0,1,1,1,1,1],
[1,0,1,8/3,x,8/3], [1,1,0,1,8/3,y], [1,8/3,1,0,1,8/3],
[1,x,8/3,1,0,1], [1,8/3,y,8/3,1,0] ]
```

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & \frac{8}{3} & x & \frac{8}{3} \\ 1 & 1 & 0 & 1 & \frac{8}{3} & y \\ 1 & \frac{8}{3} & 1 & 0 & 1 & \frac{8}{3} \\ 1 & x & \frac{8}{3} & 1 & 0 & 1 \\ 1 & \frac{8}{3} & y & \frac{8}{3} & 1 & 0 \end{bmatrix}$$

Type:
SquareMatrix(6,DistributedMultivariatePolynomial([x,y,z],Fraction Integer))

For the cyclohexan, the distances have to satisfy this equation.

```
eq := determinant mfzn
```

$$-x^2\ y^2 + \frac{22}{3}\ x^2\ y - \frac{25}{9}\ x^2 + \frac{22}{3}\ x\ y^2 - \frac{388}{9}\ x\ y -$$
$$\frac{250}{27}\ x - \frac{25}{9}\ y^2 - \frac{250}{27}\ y + \frac{14575}{81}$$

Type: DistributedMultivariatePolynomial([x,y,z],Fraction Integer)

They also must satisfy the equations given by cyclic shifts of the indeterminates.

```
groebnerFactorize [eq, eval(eq, [x,y,z], [y,z,x]), eval(eq,
[x,y,z], [z,x,y])]
```

$$
\left[\begin{array}{l}
\left[\left(y + x - \frac{22}{3}\right) z + \left(x - \frac{22}{3}\right) y - \frac{22}{3} x + \frac{121}{3}, \right. \\
\left(x^2 - \frac{22}{3} x + \frac{25}{9}\right) z + \left(x^2 - \frac{22}{3} x + \frac{25}{9}\right) y - \frac{22}{3} x^2 + \\
\frac{388}{9} x + \frac{250}{27}, \\
\left(x^2 - \frac{22}{3} x + \frac{25}{9}\right) y^2 + \left(-\frac{22}{3} x^2 + \frac{388}{9} x + \frac{250}{27}\right) y + \\
\left. \frac{25}{9} x^2 + \frac{250}{27} x - \frac{14575}{81} \right],
\end{array}\right.
$$

$$\left[z + y - \frac{21994}{5625}, y^2 - \frac{21994}{5625} y + \frac{4427}{675}, x - \frac{463}{87}\right],$$

$$\left[z^2 + \left(-\frac{1}{2} x - \frac{11}{2}\right) z - \frac{5}{6} x + \frac{265}{18}, y - x, x^2 - \frac{38}{3} x + \frac{265}{9}\right],$$

$$\left[z - \frac{25}{9}, y - \frac{11}{3}, x - \frac{11}{3}\right],$$

$$\left[z - \frac{11}{3}, y - \frac{11}{3}, x - \frac{11}{3}\right],$$

$$\left[z + \frac{5}{3}, y + \frac{5}{3}, x + \frac{5}{3}\right],$$

$$\left.\left[z - \frac{19}{3}, y + \frac{5}{3}, x + \frac{5}{3}\right]\right]$$

```
Type: List List
DistributedMultivariatePolynomial([x,y,z],Fraction Integer)
```

The union of the solutions of this list is the solution of our original problem. If we impose positivity conditions, we get two relevant ideals. One ideal is zero-dimensional, namely $x = y = z = 11/3$, and this determines the "boat" form of the cyclohexan. The other ideal is one-dimensional, which means that we have a solution space given by one parameter. This gives the "chair" form of the cyclohexan. The parameter describes the angle of the "back of the chair."

groebnerFactorize has an optional `Boolean`-valued second argument. When it is `true` partial results are displayed, since it may happen that the calculation does not terminate in a reasonable time. See the source code for `GroebnerFactorizationPackage` in **groebf.spad** for more details about the algorithms used.

9.28 Heap

The domain `Heap(S)` implements a priority queue of objects of type `S` such that the operation `extract!` removes and returns the maximum element. The implementation represents heaps as flexible arrays (see 9.23 on page 79). The representation and algorithms give complexity of $O(\log(n))$ for insertion and extractions, and $O(n)$ for construction.

Create a heap of six elements.

```
h := heap [-4,9,11,2,7,-7]
```

$$[11, 7, 9, -4, 2, -7]$$

Type: Heap Integer

Use `insert!` to add an element.

```
insert!(3,h)
```

$$[11, 7, 9, -4, 2, -7, 3]$$

Type: Heap Integer

The operation `extract!` removes and returns the maximum element.

```
extract! h
```

$$11$$

Type: PositiveInteger

The internal structure of `h` has been appropriately adjusted.

```
h
```

$$[9, 7, 3, -4, 2, -7]$$

Type: Heap Integer

Now `extract!` elements repeatedly until none are left, collecting the elements in a list.

```
[extract!(h) while not empty?(h)]
```

$$[9, 7, 3, 2, -4, -7]$$

Type: List Integer

Another way to produce the same result is by defining a `heapsort` function.

```
heapsort(x) == (empty?  x => []; cons(extract!(x),heapsort x))
```

Void

Create another sample heap.

```
h1 := heap [17,-4,9,-11,2,7,-7]
```

$$[17, 2, 9, -11, -4, 7, -7]$$

Type: Heap Integer

Apply `heapsort` to present elements in order.

```
heapsort h1
```

$$[17, 9, 7, 2, -4, -7, -11]$$

Type: List Integer

9.29 HexadecimalExpansion

All rationals have repeating hexadecimal expansions. The operation **hex** returns these expansions of type `HexadecimalExpansion`. Operations to access the individual numerals of a hexadecimal expansion can be obtained by converting the value to `RadixExpansion(16)`. More examples of expansions are available in the 9.14 on page 58, 9.3 on page 6, and 9.51 on page 184.

Issue the system command `) show HexadecimalExpansion` to display the full list of operations defined by `HexadecimalExpansion`.

This is a hexadecimal expansion of a rational number.

```
r := hex(22/7)
```

$$3.\overline{249}$$

Type: HexadecimalExpansion

Arithmetic is exact.

```
r + hex(6/7)
```

$$4$$

Type: HexadecimalExpansion

The period of the expansion can be short or long ...

```
[hex(1/i) for i in 350..354]
```

$$\left[0.0\overline{0BB3EE721A54D88}, 0.\overline{00BAB6561}, 0.00\overline{BA2E8},\right.$$

$$\left.0.\overline{00B9A7862A0FF465879D5F}, 0.0\overline{0B92143FA36F5E02E4850FE8DBD78}\right]$$

Type: List HexadecimalExpansion

or very long!

```
hex(1/1007)
```

$$0.\overline{0041149783F0BF2C7D13933192AF6980619EE345E91EC2BB9D5CC}$$
$$\overline{A5C071E40926E54E8DDAE24196C0B2F8A0AAD60DBA57F5D4C8}$$
$$\overline{536262210C74F1}$$

Type: HexadecimalExpansion

These numbers are bona fide algebraic objects.

```
p := hex(1/4)*x**2 + hex(2/3)*x + hex(4/9)
```

$$0.4\ x^2 + 0.\overline{A}\ x + 0.\overline{71C}$$

Type: Polynomial HexadecimalExpansion

```
q := D(p, x)
```

$$0.8\ x + 0.\overline{A}$$

Type: Polynomial HexadecimalExpansion

```
g := gcd(p, q)
```

$$x + 1.\overline{5}$$

Type: Polynomial HexadecimalExpansion

9.30 Integer

AXIOM provides many operations for manipulating arbitrary precision integers. In this section we will show some of those that come from `Integer` itself plus some that are implemented in other packages. More examples of using integers are in the following sections: 9.32 on page 107, 9.14 on page 58, 9.3 on page 6, 9.29 on page 96, and 9.51 on page 184.

9.30.1 Basic Functions

The size of an integer in AXIOM is only limited by the amount of computer storage you have available. The usual arithmetic operations are available.

```
2**(5678 - 4856 + 2 * 17)
```

$$\begin{array}{l}48048107704350081471815409251259243912395261398716822634738556100 \\ 88084200076308293086342527091412083743074572278211496076276922026 \\ 43343568752733498024953930242542523045817764949544214392905306388 \\ 478705146745768073877141698859815495632935288783334250628775936\end{array}$$

Type: PositiveInteger

There are a number of ways of working with the sign of an integer. Let's use this `x` as an example.

```
x := -101
```

$$-101$$

Type: Integer

First of all, there is the absolute value function.

```
abs(x)
```

$$101$$

Type: PositiveInteger

The **sign** operation returns `-1` if its argument is negative, `0` if zero and `1` if positive.

```
sign(x)
```

$$-1$$

Type: Integer

You can determine if an integer is negative in several other ways.

```
x < 0
```

true

Type: Boolean

```
x <= -1
```

```
true
```

Type: Boolean

```
negative?(x)
```

```
true
```

Type: Boolean

Similarly, you can find out if it is positive.

```
x > 0
```

```
false
```

Type: Boolean

```
x >= 1
```

```
false
```

Type: Boolean

```
positive?(x)
```

```
false
```

Type: Boolean

This is the recommended way of determining whether an integer is zero.

```
zero?(x)
```

```
false
```

Type: Boolean

> Use the **zero?** operation whenever you are testing a mathematical object for equality with zero. This is usually substantially more efficient that using "=" (as an example, think of matrices: it is easier to tell if a matrix is zero by just checking term by term than constructing another "zero" matrix and comparing the two matrices term by term) and also avoids the problem that "=" is used for creating equations.

This is the recommended way of determining whether an integer is equal to one.

```
one?(x)
```

false

Type: Boolean

This syntax is used to test equality using "=". It says that you want a `Boolean` (`true` or `false`) answer rather than an equation.

```
(x = -101)@Boolean
```

true

Type: Boolean

The operations **odd?** and **even?** determine whether an integer is odd or even, respectively. They each return a `Boolean` object.

```
odd?(x)
```

true

Type: Boolean

```
even?(x)
```

false

Type: Boolean

The operation **gcd** computes the greatest common divisor of two integers.

```
gcd(56788,43688)
```

$$4$$

Type: PositiveInteger

The operation **lcm** computes their least common multiple.

```
lcm(56788,43688)
```

$$620238536$$

Type: PositiveInteger

To determine the maximum of two integers, use **max**.

```
max(678,567)
```

$$678$$

Type: `PositiveInteger`

To determine the minimum, use **min**.

```
min(678,567)
```

$$567$$

Type: `PositiveInteger`

The `reduce` operation is used to extend binary operations to more than two arguments. For example, you can use `reduce` to find the maximum integer in a list or compute the least common multiple of all integers in the list.

```
reduce(max,[2,45,-89,78,100,-45])
```

$$100$$

Type: `PositiveInteger`

```
reduce(min,[2,45,-89,78,100,-45])
```

$$-89$$

Type: `Integer`

```
reduce(gcd,[2,45,-89,78,100,-45])
```

$$1$$

Type: `PositiveInteger`

```
reduce(lcm,[2,45,-89,78,100,-45])
```

$$1041300$$

Type: `PositiveInteger`

The infix operator "/" is *not* used to compute the quotient of integers. Rather, it is used to create rational numbers as described in 9.25 on page 89.

```
13 / 4
```

$$\frac{13}{4}$$

Type: `Fraction Integer`

The infix operation **quo** computes the integer quotient.

```
13 quo 4
```

$$3$$

Type: PositiveInteger

The infix operation **rem** computes the integer remainder.

```
13 rem 4
```

$$1$$

Type: PositiveInteger

One integer is evenly divisible by another if the remainder is zero. The operation **exquo** can also be used.

```
zero?(167604736446952 rem 2003644)
```

true

Type: Boolean

The operation **divide** returns a record of the quotient and remainder and thus is more efficient when both are needed.

```
d := divide(13,4)
```

$$[quotient = 3, remainder = 1]$$

Type: Record(quotient: Integer,remainder: Integer)

```
d.quotient
```

$$3$$

Type: PositiveInteger

```
d.remainder
```

$$1$$

Type: PositiveInteger

9.30.2 Primes and Factorization

Use the operation **factor** to factor integers. It returns an object of type `Factored Integer`. See 9.19 on page 66 for a discussion of the manipulation of factored objects.

```
factor 102400
```

$$2^{12}\ 5^2$$

Type: `Factored Integer`

The operation **prime?** returns `true` or `false` depending on whether its argument is a prime.

```
prime?  7
```

`true`

Type: `Boolean`

```
prime?  8
```

`false`

Type: `Boolean`

The operation **nextPrime** returns the least prime number greater than its argument.

```
nextPrime 100
```

$$101$$

Type: `PositiveInteger`

The operation **prevPrime** returns the greatest prime number less than its argument.

```
prevPrime 100
```

$$97$$

Type: `PositiveInteger`

To compute all primes between two integers (inclusively), use the operation **primes**.

```
primes(100,175)
```

$$[173, 167, 163, 157, 151, 149, 139, 137, 131, 127, 113, 109, 107, 103, 101]$$

`Type: List Integer`

You might sometimes want to see the factorization of an integer when it is considered a *Gaussian integer*. See 9.10 on page 34 for more details.

```
factor(2 :: Complex Integer)
```

$$-\%i\ (1 + \%i)^2$$

`Type: Factored Complex Integer`

9.30.3 Some Number Theoretic Functions

AXIOM provides several number theoretic operations for integers. More examples are in 9.32 on page 107.

The operation **fibonacci** computes the Fibonacci numbers. The algorithm has running time $O\left(\log^3(n)\right)$ for argument `n`.

```
[fibonacci(k) for k in 0..]
```

$$[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots]$$

`Type: Stream Integer`

The operation **legendre** computes the Legendre symbol for its two integer arguments where the second one is prime. If you know the second argument to be prime, use **jacobi** instead where no check is made.

```
[legendre(i,11) for i in 0..10]
```

$$[0, 1, -1, 1, 1, 1, -1, -1, -1, 1, -1]$$

`Type: List Integer`

The operation **jacobi** computes the Jacobi symbol for its two integer arguments. By convention, `0` is returned if the greatest common divisor of the numerator and denominator is not `1`.

```
[jacobi(i,15) for i in 0..9]
```

$$[0, 1, 1, 0, 1, 0, 0, -1, 1, 0]$$

`Type: List Integer`

The operation **eulerPhi** computes the values of Euler's ϕ-function where $\phi(n)$ equals the number of positive integers less than or equal to `n` that are relatively prime to the positive integer `n`.

```
[eulerPhi i for i in 1..]
```

$$[1, 1, 2, 2, 4, 2, 6, 4, 6, 4, \ldots]$$

Type: `Stream Integer`

The operation **moebiusMu** computes the Möbius μ function.

```
[moebiusMu i for i in 1..]
```

$$[1, -1, -1, 0, -1, 1, -1, 0, 0, 1, \ldots]$$

Type: `Stream Integer`

Although they have somewhat limited utility, AXIOM provides Roman numerals.

```
a := roman(78)
```

LXXVIII

Type: `RomanNumeral`

```
b := roman(87)
```

LXXXVII

Type: `RomanNumeral`

```
a + b
```

CLXV

Type: `RomanNumeral`

```
a * b
```

MMMMMMDCCLXXXVI

Type: `RomanNumeral`

```
b rem a
```

IX

Type: `RomanNumeral`

9.31 IntegerLinearDependence

The elements $v_1, \ldots, v_n$ of a module `M` over a ring `R` are said to be *linearly dependent over* `R` if there exist $c_1, \ldots, c_n$ in `R`, not all 0, such that $c_1 v_1 + \ldots c_n v_n = 0$. If such c_i's exist, they form what is called a *linear dependence relation over* `R` for the v_i's.

The package `IntegerLinearDependence` provides functions for testing whether some elements of a module over the integers are linearly dependent over the integers, and to find the linear dependence relations, if any.

Consider the domain of two by two square matrices with integer entries.

```
M := SQMATRIX(2,INT)
```

$$SquareMatrix(2, Integer)$$

Type: Domain

Now create three such matrices.

```
m1:  M := squareMatrix matrix [ [1, 2], [0, -1] ]
```

$$\begin{bmatrix} 1 & 2 \\ 0 & -1 \end{bmatrix}$$

Type: SquareMatrix(2,Integer)

```
m2:  M := squareMatrix matrix [ [2, 3], [1, -2] ]
```

$$\begin{bmatrix} 2 & 3 \\ 1 & -2 \end{bmatrix}$$

Type: SquareMatrix(2,Integer)

```
m3:  M := squareMatrix matrix [ [3, 4], [2, -3] ]
```

$$\begin{bmatrix} 3 & 4 \\ 2 & -3 \end{bmatrix}$$

Type: SquareMatrix(2,Integer)

This tells you whether `m1`, `m2` and `m3` are linearly dependent over the integers.

```
linearlyDependentOverZ? vector [m1, m2, m3]
```

true

Type: Boolean

Since they are linearly dependent, you can ask for the dependence relation.

```
c := linearDependenceOverZ vector [m1, m2, m3]
```

$$[1, -2, 1]$$

Type: Union(Vector Integer,...)

This means that the following linear combination should be 0.

```
c.1 * m1 + c.2 * m2 + c.3 * m3
```

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Type: SquareMatrix(2,Integer)

When a given set of elements are linearly dependent over R, this also means that at least one of them can be rewritten as a linear combination of the others with coefficients in the quotient field of R.

To express a given element in terms of other elements, use the operation **solveLinearlyOverQ**.

```
solveLinearlyOverQ(vector [m1, m3], m2)
```

$$\left[\frac{1}{2}, \frac{1}{2}\right]$$

Type: Union(Vector Fraction Integer,...)

9.32 IntegerNumberTheoryFunctions

The `IntegerNumberTheoryFunctions` package contains a variety of operations of interest to number theorists. Many of these operations deal with divisibility properties of integers. (Recall that an integer `a` divides an integer `b` if there is an integer `c` such that `b = a * c`.)

The operation **divisors** returns a list of the divisors of an integer.

```
div144 := divisors(144)
```

$$[1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 36, 48, 72, 144]$$

Type: List Integer

You can now compute the number of divisors of `144` and the sum of the divisors of `144` by counting and summing the elements of the list we just created.

```
#(div144)
```

$$15$$

Type: PositiveInteger

```
reduce(+,div144)
```

$$403$$

Type: PositiveInteger

Of course, you can compute the number of divisors of an integer n, usually denoted d(n), and the sum of the divisors of an integer n, usually denoted σ(n), without ever listing the divisors of n.

In AXIOM, you can simply call the operations **numberOfDivisors** and **sumOfDivisors**.

```
numberOfDivisors(144)
```

$$15$$

Type: PositiveInteger

```
sumOfDivisors(144)
```

$$403$$

Type: PositiveInteger

The key is that d(n) and σ(n) are "multiplicative functions." This means that when n and m are relatively prime, that is, when n and m have no prime factor in common, then d(nm) = d(n) d(m) and σ(nm) = σ(n) σ(m). Note that these functions are trivial to compute when n is a prime power and are computed for general n from the prime factorization of n. Other examples of multiplicative functions are σ_k(n), the sum of the k-th powers of the divisors of n and $\varphi(n)$, the number of integers between 1 and n which are prime to n. The corresponding AXIOM operations are called **sumOfKthPowerDivisors** and **eulerPhi**.

An interesting function is μ(n), the Möbius μ function, defined as follows: μ(1) = 1, μ(n) = 0, when n is divisible by a square, and $\mu = (-1)^k$, when n is the product of k distinct primes. The corresponding AXIOM operation is **moebiusMu**. This function occurs in the following theorem:

Theorem (Möbius Inversion Formula):
Let f(n) be a function on the positive integers and let F(n) be defined by

$$F(n) = \sum_{d|n} f(n)$$

where the sum is taken over the positive divisors of n. Then the values of f(n) can be recovered from the values of F(n):

$$f(n) = \sum_{d|n} \mu(n) F(\frac{n}{d})$$

where again the sum is taken over the positive divisors of n.

When f(n) = 1, then F(n) = d(n). Thus, if you sum $\mu(d) \cdot d(n/d)$ over the positive divisors d of n, you should always get 1.

```
f1(n) == reduce(+,[moebiusMu(d) * numberOfDivisors(quo(n,d)) for
d in divisors(n)])
```

Void

```
f1(200)
```

$$1$$

Type: PositiveInteger

```
f1(846)
```

$$1$$

Type: PositiveInteger

Similarly, when f(n) = n, then F(n) = σ(n). Thus, if you sum μ(d) $\cdot$ σ(n/d) over the positive divisors d of n, you should always get n.

```
f2(n) == reduce(+,[moebiusMu(d) * sumOfDivisors(quo(n,d)) for d
in divisors(n)])
```

Void

```
f2(200)
```

$$200$$

Type: PositiveInteger

```
f2(846)
```

$$846$$

Type: PositiveInteger

The Möbius inversion formula is derived from the multiplication of formal Dirichlet series. A Dirichlet series is an infinite series of the form

$$\sum_{n=1}^{\infty} a(n)n^{-s}$$

When

$$\sum_{n=1}^{\infty} a(n)n^{-s} \cdot \sum_{n=1}^{\infty} b(n)n^{-s} = \sum_{n=1}^{\infty} c(n)n^{-s}$$

then $c(n) = \sum_{d|n} a(d)b(n/d)$. Recall that the Riemann ζ function is defined by

$$\zeta(s) = \prod_{p}(1 - p^{-s})^{-1} = \sigma_{n=1}^{\infty} n^{-s}$$

where the product is taken over the set of (positive) primes. Thus,

$$\zeta(s)^{-1} = \prod_{p}(1 - p^{-s}) = \sigma_{n=1}^{\infty} n^{-s}$$

Now if $F(n) = \sum_{d|n} f(d)$, then

$$\sum f(n)n^{-s} \cdot \zeta(s) = \sum F(n)n^{-s}$$

Thus,

$$\zeta(s)^{-1} \cdot \sum F(n)n^{-s} = \sum f(n)n^{-s}$$

and $f(n) = \sum_{d|n} \mu(d)F(n/d)$.

The Fibonacci numbers are defined by $F(1) = F(2) = 1$ and $F(n) = F(n-1) + F(n-2)$ for $n = 3, 4, \ldots$.

The operation **fibonacci** computes the n-th Fibonacci number.

```
fibonacci(25)
```

$$75025$$

Type: PositiveInteger

```
[fibonacci(n) for n in 1..15]
```

$$[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]$$

Type: List Integer

Fibonacci numbers can also be expressed as sums of binomial coefficients.

```
fib(n) == reduce(+,[binomial(n-1-k,k) for k in 0..quo(n-1,2)])
```

Void

```
fib(25)
```

$$75025$$

Type: PositiveInteger

```
[fib(n) for n in 1..15]
```

$$[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]$$

Type: List Integer

Quadratic symbols can be computed with the operations **legendre** and **jacobi**. The Legendre symbol $\left(\frac{a}{p}\right)$ is defined for integers a and p with p an odd prime number. By definition, $\left(\frac{a}{p}\right) = +1$, when a is a square (mod p), $\left(\frac{a}{p}\right) =$ -1, when a is not a square (mod p), and $\left(\frac{a}{p}\right) = 0$, when a is divisible by p.

You compute $\left(\frac{a}{p}\right)$ via the command `legendre(a,p)`.

```
legendre(3,5)
```

$$-1$$

Type: Integer

```
legendre(23,691)
```

$$-1$$

Type: Integer

The Jacobi symbol $\left(\frac{a}{n}\right)$ is the usual extension of the Legendre symbol, where `n` is an arbitrary integer. The most important property of the Jacobi symbol is the following: if `K` is a quadratic field with discriminant `d` and quadratic character χ, then $\chi(n) = (d/n)$. Thus, you can use the Jacobi symbol to compute, say, the class numbers of imaginary quadratic fields from a standard class number formula.

This function computes the class number of the imaginary quadratic field with discriminant `d`.

```
h(d) == quo(reduce(+, [jacobi(d,k) for k in 1..quo(-d, 2)]), 2 -
jacobi(d,2))
```

Type: Void

```
h(-163)
```

$$1$$

Type: PositiveInteger

```
h(-499)
```

$$3$$

Type: PositiveInteger

```
h(-1832)
```

$$26$$

Type: PositiveInteger

9.33 KeyedAccessFile

The domain `KeyedAccessFile(S)` provides files which can be used as associative tables. Data values are stored in these files and can be retrieved according to their keys. The keys must be strings so this type behaves very much like the `StringTable(S)` domain. The difference is that keyed access files reside in secondary storage while string tables are kept in memory. For more information on table-oriented operations, see the description of `Table`.

Before a keyed access file can be used, it must first be opened. A new file can be created by opening it for output.

```
ey: KeyedAccessFile(Integer) := open("/tmp/editor.year",
"output")
```

"/tmp/editor.year"

Type: KeyedAccessFile Integer

Just as for vectors, tables or lists, values are saved in a keyed access file by setting elements.

```
ey."Char" := 1986
```

1986

Type: PositiveInteger

```
ey."Caviness" := 1985
```

1985

Type: PositiveInteger

```
ey."Fitch" := 1984
```

1984

Type: PositiveInteger

Values are retrieved using application, in any of its syntactic forms.

```
ey."Char"
```

1986

Type: PositiveInteger

```
ey("Char")
```

1986

Type: PositiveInteger

```
ey "Char"
```

1986

Type: PositiveInteger

Attempting to retrieve a non-existent element in this way causes an error. If it is not known whether a key exists, you should use the **search** operation.

```
search("Char", ey)
```

1986

Type: Union(Integer,...)

```
search("Smith", ey)
```

"failed"

Type: Union("failed",...)

When an entry is no longer needed, it can be removed from the file.

```
remove!("Char", ey)
```

1986

Type: Union(Integer,...)

The **keys** operation returns a list of all the keys for a given file.

```
keys ey
```

["Fitch", "Caviness"]

Type: List String

The # operation gives the number of entries.

```
#ey
```

2

Type: PositiveInteger

The table view of keyed access files provides safe operations. That is, if the AXIOM program is terminated between file operations, the file is left in a consistent, current state. This means, however, that the operations are somewhat costly. For example, after each update the file is closed.

Here we add several more items to the file, then check its contents.

```
KE := Record(key: String, entry: Integer)
```

Record(key: String,entry: Integer)

Type: Domain

```
reopen!(ey, "output")
```

"/tmp/editor.year"

Type: KeyedAccessFile Integer

If many items are to be added to a file at the same time, then it is more efficient to use the **write** operation.

```
write!(ey, ["van Hulzen", 1983]$KE)
```

$$[key = \texttt{"van Hulzen"}, entry = 1983]$$

Type: Record(key: String,entry: Integer)

```
write!(ey, ["Calmet", 1982]$KE)
```

$$[key = \texttt{"Calmet"}, entry = 1982]$$

Type: Record(key: String,entry: Integer)

```
write!(ey, ["Wang", 1981]$KE)
```

$$[key = \texttt{"Wang"}, entry = 1981]$$

Type: Record(key: String,entry: Integer)

```
close! ey
```

"/tmp/editor.year"

Type: KeyedAccessFile Integer

The **read** operation is also available from the file view, but it returns elements in a random order. It is generally clearer and more efficient to use the **keys** operation and to extract elements by key.

```
keys ey
```

$$[\texttt{"Wang"}, \texttt{"Calmet"}, \texttt{"van Hulzen"}, \texttt{"Fitch"}, \texttt{"Caviness"}]$$

Type: List String

```
members ey
```

$$[1981, 1982, 1983, 1984, 1985]$$

Type: List Integer

```
)system rm -r /tmp/editor.year
```

For more information on related topics, see 9.21 on page 74, 9.65 on page 219, and 9.34 on page 116. Issue the system command `) show KeyedAccessFile` to display the full list of operations defined by `KeyedAccessFile`.

9.34 Library

The `Library` domain provides a simple way to store AXIOM values in a file. This domain is similar to `KeyedAccessFile` but fewer declarations are needed and items of different types can be saved together in the same file.

To create a library, you supply a file name.

```
stuff := library "/tmp/Neat.stuff"
```

"/tmp/Neat.stuff"

Type: Library

Now values can be saved by key in the file. The keys should be mnemonic, just as the field names are for records. They can be given either as strings or symbols.

```
stuff.int := 32**2
```

$$1024$$

Type: PositiveInteger

```
stuff."poly" := x**2 + 1
```

$$x^2 + 1$$

Type: Polynomial Integer

```
stuff.str := "Hello"
```

"Hello"

Type: String

You obtain the set of available keys using the **keys** operation.

```
keys stuff
```

["str", "poly", "int"]

Type: List String

You extract values by giving the desired key in this way.

```
stuff.poly
```

$$x^2 + 1$$

Type: Polynomial Integer

```
stuff("poly")
```

$$x^2 + 1$$

Type: Polynomial Integer

When the file is no longer needed, you should remove it from the file system.

```
)system rm -rf /tmp/Neat.stuff
```

For more information on related topics, see 9.21 on page 74, 9.65 on page 219, and 9.33 on page 112. Issue the system command `)show Library` to display the full list of operations defined by `Library`.

9.35 LinearOrdinaryDifferentialOperator

`LinearOrdinaryDifferentialOperator(A, M)` is the domain of linear ordinary differential operators with coefficients in the differential ring `A` and operating on `M`, an `A`-module. This includes the cases of operators which are polynomials in `D` acting upon scalar or vector expressions of a single variable. The coefficients of the operator polynomials can be integers, rational functions, matrices or elements of other domains. Issue the system command `)show LinearOrdinaryDifferentialOperator` to display the full list of operations defined by `LinearOrdinaryDifferentialOperator`.

9.35.1 Differential Operators with Constant Coefficients

This example shows differential operators with rational number coefficients operating on univariate polynomials.

We begin by making type assignments so we can conveniently refer to univariate polynomials in `x` over the rationals.

```
Q := Fraction Integer
```

Fraction Integer

Type: Domain

```
PQ := UnivariatePolynomial('x, Q)
```

UnivariatePolynomial(x,Fraction Integer)

Type: Domain

```
x: PQ := 'x
```

$$x$$

Type: UnivariatePolynomial(x,Fraction Integer)

Now we assign `Dx` to be the differential operator **D** corresponding to `d/dx`.

```
Dx: LODO2(Q, PQ) := D()
```

$$D$$

Type: LinearOrdinaryDifferentialOperator(Fraction Integer, UnivariatePolynomial(x,Fraction Integer))

New operators are created as polynomials in `D()`.

```
a := Dx + 1
```

$$D + 1$$

Type: LinearOrdinaryDifferentialOperator(Fraction Integer, UnivariatePolynomial(x,Fraction Integer))

```
b := a + 1/2*Dx**2 - 1/2
```

$$\frac{1}{2}\ D^2 + D + \frac{1}{2}$$

Type: LinearOrdinaryDifferentialOperator(Fraction Integer, UnivariatePolynomial(x,Fraction Integer))

To apply the operator `a` to the value `p` the usual function call syntax is used.

```
p := 4*x**2 + 2/3
```

$$4\ x^2 + \frac{2}{3}$$

Type: UnivariatePolynomial(x,Fraction Integer)

```
a p
```

$$4\ x^2 + 8\ x + \frac{2}{3}$$

Type: UnivariatePolynomial(x,Fraction Integer)

Operator multiplication is defined by the identity `(a*b) p = a(b(p))`

```
(a * b) p = a b p
```

$$2\ x^2 + 12\ x + \frac{37}{3} = 2\ x^2 + 12\ x + \frac{37}{3}$$

Type: Equation UnivariatePolynomial(x,Fraction Integer)

Exponentiation follows from multiplication.

```
c := (1/9)*b*(a + b)**2
```

$$\frac{1}{72}\ D^6 + \frac{5}{36}\ D^5 + \frac{13}{24}\ D^4 + \frac{19}{18}\ D^3 + \frac{79}{72}\ D^2 + \frac{7}{12}\ D + \frac{1}{8}$$

Type: LinearOrdinaryDifferentialOperator(Fraction Integer, UnivariatePolynomial(x,Fraction Integer))

Finally, note that operator expressions may be applied directly.

```
(a**2 - 3/4*b + c) (p + 1)
```

$$3\ x^2 + \frac{44}{3}\ x + \frac{541}{36}$$

Type: UnivariatePolynomial(x,Fraction Integer)

9.35.2 Differential Operators with Rational Function Coefficients

This example shows differential operators with rational function coefficients. In this case operator multiplication is non-commutative and, since the coefficients form a field, an operator division algorithm exists.

We begin by defining `RFZ` to be the rational functions in `x` with integer coefficients and `Dx` to be the differential operator for `d/dx`.

```
RFZ := Fraction UnivariatePolynomial('x, Integer)
```

Fraction UnivariatePolynomial(x,Integer)

Type: Domain

```
x : RFZ := 'x
```

$$x$$

Type: Fraction UnivariatePolynomial(x,Integer)

```
Dx : LODO RFZ := D()
```

$$D$$

Type: LinearOrdinaryDifferentialOperator Fraction UnivariatePolynomial(x,Integer)

Operators are created using the usual arithmetic operations.

```
b := 3*x**2*Dx**2 + 2*Dx + 1/x
```

$$3\ x^2\ D^2 + 2\ D + \frac{1}{x}$$

Type: LinearOrdinaryDifferentialOperator Fraction UnivariatePolynomial(x,Integer)

```
a := b*(5*x*Dx + 7)
```

$$15\ x^3\ D^3 + \left(51\ x^2 + 10\ x\right)\ D^2 + 29\ D + \frac{7}{x}$$

Type: LinearOrdinaryDifferentialOperator Fraction UnivariatePolynomial(x,Integer)

Operator multiplication corresponds to functional composition.

```
p := x**2 + 1/x**2
```

$$\frac{x^4 + 1}{x^2}$$

Type: Fraction UnivariatePolynomial(x,Integer)

Since operator coefficients depend on x, the multiplication is not commutative.

```
(a*b) p = a(b(p))
```

$$\frac{612x^6 + 510x^5 + 180x^4 - 972x^2 + 1026x - 120}{x^4} = \frac{612x^6 + 510x^5 + 180x^4 - 972x^2 + 1026x - 120}{x^4}$$

Type: Equation Fraction UnivariatePolynomial(x, Integer)

```
(b*a) p = b(a(p))
```

$$\frac{612x^6 + 510x^5 + 255x^4 - 972x^2 + 486x - 45}{x^4} = \frac{612x^6 + 510x^5 + 255x^4 - 972x^2 + 486x - 45}{x^4}$$

Type: Equation Fraction UnivariatePolynomial(x,Integer)

When the coefficients of operator polynomials come from a field, as in this case, it is possible to define operator division. Division on the left and division on the right yield different results when the multiplication is non-commutative.

The results of **leftDivide** and **rightDivide** are quotient-remainder pairs satisfying:

`leftDivide(a,b) = [q, r]` such that `a = b*q + r`
`rightDivide(a,b) = [q, r]` such that `a = q*b + r`

In both cases, the **degree** of the remainder, `r`, is less than the degree of `b`.

```
ld := leftDivide(a,b)
```

$$[quotient = 5\ x\ D + 7, remainder = 0]$$

Type: Record(quotient: LinearOrdinaryDifferentialOperator1 Fraction UnivariatePolynomial(x,Integer), remainder: LinearOrdinaryDifferentialOperator1 Fraction UnivariatePolynomial(x,Integer))

```
a = b * ld.quotient + ld.remainder
```

$$15\ x^3\ D^3 + \left(51\ x^2 + 10\ x\right)\ D^2 + 29\ D + \tfrac{7}{x} =$$

$$15\ x^3\ D^3 + \left(51\ x^2 + 10\ x\right)\ D^2 + 29\ D + \frac{7}{x}$$

Type: Equation LinearOrdinaryDifferentialOperator Fraction UnivariatePolynomial(x,Integer)

The operations of left and right division are so-called because the quotient is obtained by dividing `a` on that side by `b`.

```
rd := rightDivide(a,b)
```

$$\left[quotient = 5\ x\ D + 7, remainder = 10\ D + \frac{5}{x}\right]$$

Type: Record(quotient: LinearOrdinaryDifferentialOperator Fraction UnivariatePolynomial(x,Integer), remainder: LinearOrdinaryDifferentialOperator Fraction UnivariatePolynomial(x,Integer))

```
a = rd.quotient * b + rd.remainder
```

$$15\ x^3\ D^3 + \left(51\ x^2 + 10\ x\right)\ D^2 + 29\ D + \tfrac{7}{x} =$$

$$15\ x^3\ D^3 + \left(51\ x^2 + 10\ x\right)\ D^2 + 29\ D + \frac{7}{x}$$

Type: Equation LinearOrdinaryDifferentialOperator Fraction UnivariatePolynomial(x,Integer)

Operations **rightQuotient** and **rightRemainder** are available if only one of the quotient or remainder are of interest to you. This is the quotient from right division.

```
rightQuotient(a,b)
```

$$5\ x\ D + 7$$

Type: LinearOrdinaryDifferentialOperator Fraction UnivariatePolynomial(x,Integer)

This is the remainder from right division. The corresponding "left" functions **leftQuotient** and **leftRemainder** are also available.

```
rightRemainder(a,b)
```

$$10\ D + \frac{5}{x}$$

Type: LinearOrdinaryDifferentialOperator Fraction UnivariatePolynomial(x,Integer)

For exact division, the operations **leftExactQuotient** and **rightExact Quotient** are supplied. These return the quotient but only if the remainder is zero. The call `rightExactQuotient(a,b)` would yield an error.

```
leftExactQuotient(a,b)
```

$$5\ x\ D + 7$$

Type: Union(LinearOrdinaryDifferentialOperator Fraction UnivariatePolynomial(x,Integer),...)

The division operations allow the computation of left and right greatest common divisors (**leftGcd** and **rightGcd**) via remainder sequences, and consequently the computation of left and right least common multiples (**rightLcm** and **leftLcm**).

```
e := leftGcd(a,b)
```

$$3\ x^2\ D^2 + 2\ D + \frac{1}{x}$$

Type: LinearOrdinaryDifferentialOperator Fraction UnivariatePolynomial(x,Integer)

Note that a greatest common divisor doesn't necessarily divide `a` and `b` on both sides. Here the left greatest common divisor does not divide `a` on the right.

```
leftRemainder(a, e)
```

$$0$$

Type: LinearOrdinaryDifferentialOperator Fraction UnivariatePolynomial(x,Integer)

```
rightRemainder(a, e)
```

$$10\ D + \frac{5}{x}$$

Type: LinearOrdinaryDifferentialOperator Fraction UnivariatePolynomial(x,Integer)

Similarly, a least common multiple is not necessarily divisible from both sides.

```
f := rightLcm(a,b)
```

$$20x^5 D^5 + \frac{684x^4 + 80x^3}{3} D^4 + \frac{5832x^3 + 1656x^2 + 80x}{9} D^3 + \frac{3672x^2 + 2040x + 352}{9} D^2 + \frac{172}{9x} D - \frac{28}{9x^2}$$

Type: LinearOrdinaryDifferentialOperator(Fraction UnivariatePolynomial(x,Integer),(Fraction UnivariatePolynomial(x,Integer))

```
rightRemainder(f, b)
```

$$0$$

Type: LinearOrdinaryDifferentialOperator(Fraction UnivariatePolynomial(x,Integer),(Fraction UnivariatePolynomial(x,Integer))

```
leftRemainder(f, b)
```

$$\frac{-1176x + 160}{9x} D + \frac{312x - 80}{9x^2}$$

Type: LinearOrdinaryDifferentialOperator(Fraction UnivariatePolynomial(x,Integer),(Fraction UnivariatePolynomial(x,Integer))

9.35.3 Differential Operators with Series Coefficients

Problem: Find the first few coefficients in `x` of `L3 phi` where

```
L3 = (d/dx)**3 + G*x**2 * d/dx + H*x**3 - exp(x)
```

```
phi = sum s[i]*x**i for i = 0..
```

We work with Taylor series in `x`.
Solution:

```
T := UnivariateTaylorSeries(Expression Integer , 'x,0)
```

UnivariateTaylorSeries(Expression Integer , 'x,0)

Type: Domain

```
x: T: 'x
```

$$x$$

Type: UnivariateTaylorSeries(Expression Integer , 'x,0)

Define the operator `L3` and the series `phi` with undetermined coefficents.

```
Dx: LODO(T,T) := D()
```

$$D$$

Type: LinearOrdinaryDifferentialOperator

(UnivariateTaylorSeries(ExpressionInteger,x,0),

UnivariateTaylorSeries(ExpressionInteger,x,0))

```
L3 := Dx**3 + G * x**2 * Dx + x**3 * H - exp(x)
```

$$D^3+Gx^2D-1-x-\frac{1}{2}x^2+\frac{6H-1}{6}x^3-\frac{1}{24}x^4-\frac{1}{120}x^5-\frac{1}{720}x^6-\frac{1}{5040}x^7+O(x^8)$$

Type: LinearOrdinaryDifferentialOperator

(UnivariateTaylorSeries(ExpressionInteger,x,0),

UnivariateTaylorSeries(ExpressionInteger,x,0))

```
s: Symbol := 's
```

$$s$$

Type: Symbol

```
phi: T := series([s[i] for i in 0..])
```

$$s_0 + s_1 x + s_2 x^2 + s_3 x^3 + s_4 x^4 + s_5 x^5 + s_6 x^6 + s_7 x^7 + O(x^8)$$

UnivariateTaylorSeries(ExpressionInteger,x,0))

Apply the operator to get the solution

```
L3 phi
```

$$6s_3 - s_0 + (24s_4 - s_1 - s_0)x+$$

$$\frac{120s_5 - 2s_2 + (2G-2)s_1 - s_0}{2}x^2+$$

$$\frac{720s_6 - 6s_3 + (12G-6)s_2 - 3s_1 + (6H-1)s_0}{6}x^3+$$

$$\frac{\left(\begin{array}{l} 5040s_7 - 24s_4 + (72G-24)s_3 - 12s_2+ \\ (24H-4)s_1 - s_0 \end{array}\right)}{24}x^4+$$

$$\frac{\left(\begin{array}{l} 40320s_8 - 120s_5 + (480G-120)s_4 - 60s_3+ \\ (120H-20)s_2 - 5s_1 - s_0 \end{array}\right)}{120}x^5+$$

$$\frac{\left(\begin{array}{l} 362880s_9 - 720s_6 + (3600G-720)s_5 - 360s_4+ \\ (720H-120)s_3 - 30s_2 - 6s_1 - s_0 \end{array}\right)}{720}x^6+$$

$$\frac{\left(\begin{array}{l} 3628800s_{10} - 5040s_7 + (30240G-5040)s_6- \\ 2520s_5 + (5040H-840)s_4 - 210s_3 - 42s_2 - 7s_1- \\ s_0 \end{array}\right)}{5040}x^7+$$

$$O(x^8)$$

9.35.4 Differential Operators with Matrix Coefficients Operating on Vectors

This is another example of linear ordinary differential operators with non-commutative multiplication. Unlike the rational function case, the differential ring of square matrices (of a given dimension) with univariate polynomial entries does not form a field. Thus the number of operations available is more limited.

In this section, the operators have three by three matrix coefficients with polynomial entries.

```
PZ := UnivariatePolynomial(x,Integer)
```

$$UnivariatePolynomial(x, Integer)$$

Type: Domain

```
x:PZ := 'x
```

$$x$$

Type: UnivariatePolynomial(x,Integer)

```
Mat := SquareMatrix(3,PZ)
```

$$SquareMatrix(3, UnivariatePolynomial(x, Integer))$$

Type: Domain

The operators act on the vectors considered as a `Mat`-module.

```
Vect := DPMM(3, PZ, Mat, PZ)
```

Type: Domain

```
Modo := LODO(Mat, Vect)
```

Type: Domain

The matrix `m` is used as a coefficient and the vectors `p` and `q` are operated upon.

```
m:Mat := matrix [ [x**2,1,0],[1,x**4,0],[0,0,4*x**2] ]
```

$$\begin{bmatrix} x^2 & 1 & 0 \\ 1 & x^4 & 0 \\ 0 & 0 & 4\,x^2 \end{bmatrix}$$

Type: SquareMatrix(3,UnivariatePolynomial(x,Integer))

```
p:Vect := directProduct [3*x**2+1,2*x,7*x**3+2*x]
```

$$\left[3\ x^{2}+1, 2\ x, 7\ x^{3}+2\ x\right]$$

Type: DirectProductMatrixModule(3,
UnivariatePolynomial(x,Integer),
SquareMatrix(3,UnivariatePolynomial(x,Integer)),
UnivariatePolynomial(x,Integer))

```
q: Vect := m * p
```

$$\left[3\ x^{4}+x^{2}+2\ x, 2\ x^{5}+3\ x^{2}+1, 28\ x^{5}+8\ x^{3}\right]$$

Type: DirectProductMatrixModule(3,
UnivariatePolynomial(x,Integer),
SquareMatrix(3,UnivariatePolynomial(x,Integer)),
UnivariatePolynomial(x,Integer))

Now form a few operators.

```
Dx : Modo := D()
```

$$D$$

Type: LinearOrdinaryDifferentialOperator2(
SquareMatrix(3,UnivariatePolynomial(x,Integer)),
DirectProductMatrixModule(3, UnivariatePolynomial(x,Integer),
SquareMatrix(3,UnivariatePolynomial(x,Integer)),
UnivariatePolynomial(x,Integer)))

```
a : Modo := Dx + m
```

$$D+\left[\begin{array}{ccc} x^{2} & 1 & 0 \\ 1 & x^{4} & 0 \\ 0 & 0 & 4\ x^{2} \end{array}\right]$$

Type: LinearOrdinaryDifferentialOperator2(
SquareMatrix(3,UnivariatePolynomial(x,Integer)),
DirectProductMatrixModule(3, UnivariatePolynomial(x,Integer),
SquareMatrix(3, UnivariatePolynomial(x,Integer)),
UnivariatePolynomial(x,Integer)))

```
b : Modo := m*Dx + 1
```

$$\begin{bmatrix} x^2 & 1 & 0 \\ 1 & x^4 & 0 \\ 0 & 0 & 4\,x^2 \end{bmatrix} D + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
Type: LinearOrdinaryDifferentialOperator2( SquareMatrix(3,
UnivariatePolynomial(x,Integer)), DirectProductMatrixModule(3,
UnivariatePolynomial(x,Integer), SquareMatrix(3,
UnivariatePolynomial(x,Integer)),
UnivariatePolynomial(x,Integer)))
```

```
c := a*b
```

$$\begin{bmatrix} x^2 & 1 & 0 \\ 1 & x^4 & 0 \\ 0 & 0 & 4\,x^2 \end{bmatrix} D^2 +$$

$$\begin{bmatrix} x^4 + 2\,x + 2 & x^4 + x^2 & 0 \\ x^4 + x^2 & x^8 + 4\,x^3 + 2 & 0 \\ 0 & 0 & 16\,x^4 + 8\,x + 1 \end{bmatrix} D +$$

$$\begin{bmatrix} x^2 & 1 & 0 \\ 1 & x^4 & 0 \\ 0 & 0 & 4\,x^2 \end{bmatrix}$$

```
Type: LinearOrdinaryDifferentialOperator2( SquareMatrix(3,
UnivariatePolynomial(x,Integer)), DirectProductMatrixModule(3,
UnivariatePolynomial(x,Integer), SquareMatrix(3,
UnivariatePolynomial(x,Integer)),
UnivariatePolynomial(x,Integer)))
```

These operators can be applied to vector values.

```
a p
```

$$\left[3\,x^4 + x^2 + 8\,x, 2\,x^5 + 3\,x^2 + 3, 28\,x^5 + 8\,x^3 + 21\,x^2 + 2\right]$$

```
Type: DirectProductMatrixModule(3,
UnivariatePolynomial(x,Integer), SquareMatrix(3,
UnivariatePolynomial(x,Integer)),
UnivariatePolynomial(x,Integer))
```

```
b p
```

$$\left[6\,x^3 + 3\,x^2 + 3, 2\,x^4 + 8\,x, 84\,x^4 + 7\,x^3 + 8\,x^2 + 2\,x\right]$$

```
Type: DirectProductMatrixModule(3,
UnivariatePolynomial(x,Integer), SquareMatrix(3,
UnivariatePolynomial(x,Integer)),
UnivariatePolynomial(x,Integer))
```

```
(a + b + c) (p + q)
```

$$\left[10\ x^8+12\ x^7+16\ x^6+30\ x^5+85\ x^4+94\ x^3+40\ x^2+40\ x+17,\right.$$

$$10\ x^{12}+10\ x^9+12\ x^8+92\ x^7+6\ x^6+32\ x^5+72\ x^4+28\ x^3+49\ x^2+$$
$$32\ x+19,$$

$$\left.2240\ x^8+224\ x^7+1280\ x^6+3508\ x^5+492\ x^4+751\ x^3+98\ x^2+18\ x+4\right]$$

Type: DirectProductMatrixModule(3, UnivariatePolynomial(x,Integer), SquareMatrix(3, UnivariatePolynomial(x,Integer)), UnivariatePolynomial(x,Integer))

9.36 List

A is a finite collection of elements in a specified order that can contain duplicates. A list is a convenient structure to work with because it is easy to add or remove elements and the length need not be constant. There are many different kinds of lists in AXIOM, but the default types (and those used most often) are created by the `List` constructor. For example, there are objects of type `List Integer`, `List Float` and `List Polynomial Fraction Integer`. Indeed, you can even have `List List List Boolean` (that is, lists of lists of lists of Boolean values). You can have lists of any type of AXIOM object.

9.36.1 Creating Lists

The easiest way to create a list with, for example, the elements `2, 4, 5, 6` is to enclose the elements with square brackets and separate the elements with commas.

The spaces after the commas are optional, but they do improve the readability.

```
[2, 4, 5, 6]
```

$$[2,4,5,6]$$

Type: List PositiveInteger

To create a list with the single element `1`, you can use either `[1]` or the operation **list**.

```
[1]
```

$$[1]$$

Type: List PositiveInteger

```
list(1)
```

$$[1]$$

Type: List PositiveInteger

Once created, two lists `k` and `m` can be concatenated by issuing `append(k,m)`. **append** does *not* physically join the lists, but rather produces a new list with the elements coming from the two arguments.

```
append([1,2,3],[5,6,7])
```

$$[1, 2, 3, 5, 6, 7]$$

Type: List PositiveInteger

Use **cons** to append an element onto the front of a list.

```
cons(10,[9,8,7])
```

$$[10, 9, 8, 7]$$

Type: List PositiveInteger

9.36.2 Accessing List Elements

To determine whether a list has any elements, use the operation **empty?**.

```
empty?  [x+1]
```

false

Type: Boolean

Alternatively, equality with the list constant **nil** can be tested.

```
([] = nil)@Boolean
```

true

Type: Boolean

We'll use this in some of the following examples.

```
k := [4,3,7,3,8,5,9,2]
```

$$[4, 3, 7, 3, 8, 5, 9, 2]$$

Type: List PositiveInteger

Each of the next four expressions extracts the **first** element of `k`.

```
first k
```

$$4$$

Type: PositiveInteger

```
k.first
```

$$4$$

Type: PositiveInteger

```
k.1
```

$$4$$

Type: PositiveInteger

```
k(1)
```

$$4$$

Type: PositiveInteger

The last two forms generalize to `k.i` and `k(i)`, respectively, where $1 \leq i \leq n$ and `n` equals the length of `k`.

This length is calculated by "`#`".

```
n := #k
```

$$8$$

Type: PositiveInteger

Performing an operation such as `k.i` is sometimes referred to as *indexing into k* or *elting into k*. The latter phrase comes about because the name of the operation that extracts elements is called **elt**. That is, `k.3` is just alternative syntax for `elt(k,3)`. It is important to remember that list indices begin with 1. If we issue `k := [1,3,2,9,5]` then `k.4` returns `9`. It is an error to use an index that is not in the range from `1` to the length of the list.

The last element of a list is extracted by any of the following three expressions.

```
last k
```

$$2$$

Type: PositiveInteger

```
k.last
```

$$2$$

Type: PositiveInteger

This form computes the index of the last element and then extracts the element from the list.

```
k.(#k)
```

$$2$$

Type: PositiveInteger

9.36.3 Changing List Elements

We'll use this in some of the following examples.

```
k := [4,3,7,3,8,5,9,2]
```

$$[4, 3, 7, 3, 8, 5, 9, 2]$$

Type: List PositiveInteger

List elements are reset by using the `k.i` form on the left-hand side of an assignment. This expression resets the first element of `k` to `999`.

```
k.1 := 999
```

$$999$$

Type: PositiveInteger

As with indexing into a list, it is an error to use an index that is not within the proper bounds. Here you see that `k` was modified.

```
k
```

$$[999, 3, 7, 3, 8, 5, 9, 2]$$

Type: List PositiveInteger

The operation that performs the assignment of an element to a particular position in a list is called **setelt**. This operation is *destructive* in that it changes the list. In the above example, the assignment returned the value `999` and `k` was modified. For this reason, lists are called objects: it is possible to change part of a list (mutate it) rather than always returning a new list reflecting the intended modifications.

Moreover, since lists can share structure, changes to one list can sometimes affect others.

```
k := [1,2]
```

$$[1, 2]$$

Type: List PositiveInteger

```
m := cons(0,k)
```

$$[0, 1, 2]$$

Type: List Integer

Change the second element of `m`.

```
m.2 := 99
```

$$99$$

Type: PositiveInteger

See, `m` was altered.

```
m
```

$$[0, 99, 2]$$

Type: List Integer

But what about `k`? It changed too!

```
k
```

$$[99, 2]$$

Type: List PositiveInteger

9.36.4 Other Functions

An operation that is used frequently in list processing is that which returns all elements in a list after the first element.

```
k := [1,2,3]
```

$$[1, 2, 3]$$

Type: List PositiveInteger

Use the **rest** operation to do this.

```
rest k
```

$$[2, 3]$$

Type: List PositiveInteger

To remove duplicate elements in a list `k`, use **removeDuplicates**.

```
removeDuplicates [4,3,4,3,5,3,4]
```

$$[4, 3, 5]$$

Type: List PositiveInteger

To get a list with elements in the order opposite to those in a list `k`, use **reverse**.

```
reverse [1,2,3,4,5,6]
```

$$[6, 5, 4, 3, 2, 1]$$

Type: List PositiveInteger

To test whether an element is in a list, use **member?**: `member?(a,k)` returns `true` or `false` depending on whether `a` is in `k` or not.

```
member?(1/2,[3/4,5/6,1/2])
```

true

Type: Boolean

```
member?(1/12,[3/4,5/6,1/2])
```

false

Type: Boolean

As an exercise, the reader should determine how to get a list containing all but the last of the elements in a given non-empty list `k`.[4]

[4] `reverse(rest(reverse(k)))` works.

9.36.5 Dot, Dot

Certain lists are used so often that AXIOM provides an easy way of constructing them. If `n` and `m` are integers, then `expand [n..m]` creates a list containing `n, n+1, ... m`. If `n > m` then the list is empty. It is actually permissible to leave off the `m` in the dot-dot construction (see below).

The dot-dot notation can be used more than once in a list construction and with specific elements being given. Items separated by dots are called *segments.*

```
[1..3,10,20..23]
```

$$[1..3, 10..10, 20..23]$$

Type: List Segment PositiveInteger

Segments can be expanded into the range of items between the endpoints by using **expand**.

```
expand [1..3,10,20..23]
```

$$[1, 2, 3, 10, 20, 21, 22, 23]$$

Type: List Integer

What happens if we leave off a number on the right-hand side of "`..`"?

```
expand [1..]
```

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots]$$

Type: Stream Integer

What is created in this case is a `Stream` which is a generalization of a list. See 9.60 on page 202 for more information.

9.37 MakeFunction

It is sometimes useful to be able to define a function given by the result of a calculation.

Suppose that you have obtained the following expression after several computations and that you now want to tabulate the numerical values of `f` for `x` between `-1` and `+1` with increment `0.1`.

```
expr := (x - exp x + 1)**2 * (sin(x**2) * x + 1)**3
```

$$\left(x^3\ \%e^{x^2} + \left(-2\ x^4 - 2\ x^3\right)\ \%e^x + x^5 + 2\ x^4 + x^3\right)\ \sin\left(x^2\right)^3 +$$
$$\left(3\ x^2\ \%e^{x^2} + \left(-6\ x^3 - 6\ x^2\right)\ \%e^x + 3\ x^4 + 6\ x^3 + 3\ x^2\right)\ \sin\left(x^2\right)^2 +$$
$$\left(3\ x\ \%e^{x^2} + \left(-6\ x^2 - 6\ x\right)\ \%e^x + 3\ x^3 + 6\ x^2 + 3\ x\right)\ \sin\left(x^2\right) + \%e^{x^2} +$$
$$(-2\ x - 2)\ \%e^x + x^2 + 2\ x + 1$$

Type: Expression Integer

You could, of course, use the function **eval** within a loop and evaluate `expr` twenty-one times, but this would be quite slow. A better way is to create a numerical function `f` such that `f(x)` is defined by the expression `expr` above, but without retyping `expr`! The package `MakeFunction` provides the operation **function** which does exactly this.

Issue this to create the function `f(x)` given by `expr`.

```
function(expr, f, x)
```

$$f$$

Type: Symbol

To tabulate `expr`, we can now quickly evaluate `f` 21 times.

```
tbl := [f(0.1 * i - 1) for i in 0..20];
```

```
Compiling function f with type Float -> Float
```

Type: List Float

Use the list `[x1,...,xn]` as the third argument to **function** to create a multivariate function `f(x1,...,xn)`.

```
e := (x - y + 1)**2 * (x**2 * y + 1)**2
```

$$x^4\ y^4 + \left(-2\ x^5 - 2\ x^4 + 2\ x^2\right)\ y^3 + \left(x^6 + 2\ x^5 + x^4 - 4\ x^3 - 4\ x^2 + 1\right)\ y^2 +$$
$$\left(2\ x^4 + 4\ x^3 + 2\ x^2 - 2\ x - 2\right)\ y + x^2 + 2\ x + 1$$

Type: Polynomial Integer

```
function(e, g, [x, y])
```

$$g$$

Type: Symbol

In the case of just two variables, they can be given as arguments without making them into a list.

```
function(e, h, x, y)
```

$$h$$

Type: Symbol

Note that the functions created by **function** are not limited to floating point numbers, but can be applied to any type for which they are defined.

```
m1 := squareMatrix [ [1, 2], [3, 4] ]
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Type: SquareMatrix(2,Integer)

```
m2 := squareMatrix [ [1, 0], [-1, 1] ]
```

$$\begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

Type: SquareMatrix(2,Integer)

```
h(m1, m2)
```

```
Compiling function h with type(SquareMatrix(2,Integer), squareMat-
rix(2,Integer)) -> SquareMatrix(2,Integer)
```

$$\begin{bmatrix} -7836 & 8960 \\ -17132 & 19588 \end{bmatrix}$$

Type: SquareMatrix(2,Integer)

Issue the system command `)show MakeFunction` to display the full list of operations defined by `MakeFunction`.

9.38 MappingPackage1

Function are objects of type `Mapping`. In this section we demonstrate some library operations from the packages `MappingPackage1`, `MappingPackage2`, and `MappingPackage3` that manipulate and create functions. Some terminology: a *nullary* function takes no arguments, a *unary* function takes one argument, and a *binary* function takes two arguments.

We begin by creating an example function that raises a rational number to an integer exponent.

```
power(q:  FRAC INT, n:  INT): FRAC INT == q**n
```

```
Function declaration power : (Fraction Integer,Integer) ->
   Fraction Integer has been added to workspace.
```

Type : Void

```
power(2,3)
```

```
Compiling function power with type (Fraction Integer,Integer) ->
   Fraction Integer
```

$$8$$

Type: Fraction Integer

The **twist** operation transposes the arguments of a binary function. Here `rewop(a, b)` is `power(b, a)`.

```
rewop := twist power
```

theMap(...)

Type: ((Integer,Fraction Integer) -> Fraction Integer)

This is 2^3.

```
rewop(3, 2)
```

$$8$$

Type: Fraction Integer

Now we define `square` in terms of `power`.

```
square:  FRAC INT -> FRAC INT
```

Type: Void

The **curryRight** operation creates a unary function from a binary one by providing a constant argument on the right.

```
square:= curryRight(power, 2)
```

theMap(...)

Type: (Fraction Integer -> Fraction Integer)

Likewise, the **curryLeft** operation provides a constant argument on the left.

```
square 4
```

$$16$$

Type: Fraction Integer

The **constantRight** operation creates (in a trivial way) a binary function from a unary one: `constantRight(f)` is the function `g` such that `g(a,b)= f(a)`.

```
squirrel:= constantRight(square)$MAPPKG3(FRAC INT,FRAC INT,FRAC
INT)
```

theMap(...)

Type: ((Fraction Integer,Fraction Integer) -> Fraction Integer)

Likewise, `constantLeft(f)` is the function `g` such that `g(a,b)= f(b)`.

```
squirrel(1/2, 1/3)
```

$$\frac{1}{4}$$

Type: Fraction Integer

The **curry** operation makes a unary function nullary.

```
sixteen := curry(square, 4/1)
```

theMap(...)

Type: (() -> Fraction Integer)

```
sixteen()
```

$$16$$

`Type: Fraction Integer`

The "`*`" operation constructs composed functions.

```
square2:=square*square
```

theMap(...)

`Type: (Fraction Integer -> Fraction Integer)`

```
square2 3
```

$$81$$

`Type: Fraction Integer`

Use the "`**`" operation to create functions that are `n`-fold iterations of other functions.

```
sc(x:  FRAC INT): FRAC INT == x + 1
```

```
Function declaration sc : Fraction Integer ->
   Fraction Integer has been added to workspace.
```

`Type: Void`

This is a list of `Mapping` objects.

```
incfns := [sc**i for i in 0..10]
```

[theMap(...), theMap(...), theMap(...), theMap(...), theMap(...), theMap(...), theMap(...), theMap(...), theMap(...), theMap(...), theMap(...)]

`Type: List (Fraction Integer -> Fraction Integer)`

This is a list of applications of those functions.

```
[f 4 for f in incfns]
```

$$[4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]$$

`Type: List Fraction Integer`

Use the **recur** operation for recursion:
`g := recur f` means `g(n,x) == f(n,f(n-1,...f(1,x)))`.

```
times(n:NNI, i:INT):INT == n*i
```

```
Function declaration times : (NonNegativeInteger,Integer) ->
   Integer has been added to workspace.
```

Type: Void

```
r := recur(times)
```

```
Compiling function times with type(NonNegativeInteger, Integer)->
Integer
```

theMap(...)

Type: ((NonNegativeInteger,Integer) -> Integer)

This is a factorial function.

```
fact := curryRight(r, 1)
```

theMap(...)

Type: (NonNegativeInteger -> Integer)

```
fact 4
```

24

Type: PositiveInteger

Constructed functions can be used within other functions.

```
mto2ton(m, n) ==
  raiser := square**n
  raiser m
```

Type: Void

This is 3^{2^3}.

```
mto2ton(3, 3)
```

```
   Compiling function mto2ton with type (PositiveInteger,
      PositiveInteger) -> Fraction Integer
```

6561

Type: Fraction Integer

Here `shiftfib` is a unary function that modifies its argument.

```
shiftfib(r: List INT) : INT ==
  t := r.1
  r.1 := r.2
  r.2 := r.2 + t
  t
```

```
Function declaration shiftfib : List Integer -> Integer
   has been added to workspace.
```

Type: Void

By currying over the argument we get a function with private state.

```
fibinit:  List INT := [0, 1]
```

$$[0, 1]$$

Type: List Integer

```
fibs := curry(shiftfib, fibinit)
```

```
Compiling function shiftlib with type List Integer -> Integer
```

$$\mathrm{theMap}(...)$$

Type: (() -> Integer)

```
[fibs() for i in 0..30]
```

$$[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040]$$

Type: List Integer

9.39 Matrix

The `Matrix` domain provides arithmetic operations on matrices and standard functions from linear algebra. This domain is similar to the `TwoDimensional Array` domain, except that the entries for `Matrix` must belong to a `Ring`.

9.39.1 Creating Matrices

There are many ways to create a matrix from a collection of values or from existing matrices.

If the matrix has almost all items equal to the same value, use **new** to create a matrix filled with that value and then reset the entries that are different.

```
m :  Matrix(Integer) := new(3,3,0)
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

`Type: Matrix Integer`

To change the entry in the second row, third column to `5`, use **setelt**.

```
setelt(m,2,3,5)
```

$$5$$

`Type: PositiveInteger`

An alternative syntax is to use assignment.

```
m(1,2) := 10
```

$$10$$

`Type: PositiveInteger`

The matrix was *destructively modified*.

```
m
```

$$\begin{bmatrix} 0 & 10 & 0 \\ 0 & 0 & 5 \\ 0 & 0 & 0 \end{bmatrix}$$

`Type: Matrix Integer`

If you already have the matrix entries as a list of lists, use **matrix**.

```
matrix [ [1,2,3,4],[0,9,8,7] ]
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 9 & 8 & 7 \end{bmatrix}$$

`Type: Matrix Integer`

If the matrix is diagonal, use **diagonalMatrix**.

```
dm := diagonalMatrix [1,x**2,x**3,x**4,x**5]
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & x^2 & 0 & 0 & 0 \\ 0 & 0 & x^3 & 0 & 0 \\ 0 & 0 & 0 & x^4 & 0 \\ 0 & 0 & 0 & 0 & x^5 \end{bmatrix}$$

`Type: Matrix Polynomial Integer`

Use **setRow** and **setColumn** to change a row or column of a matrix.

```
setRow!(dm,5,vector [1,1,1,1,1])
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & x^2 & 0 & 0 & 0 \\ 0 & 0 & x^3 & 0 & 0 \\ 0 & 0 & 0 & x^4 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

`Type: Matrix Polynomial Integer`

```
setColumn!(dm,2,vector [y,y,y,y,y])
```

$$\begin{bmatrix} 1 & y & 0 & 0 & 0 \\ 0 & y & 0 & 0 & 0 \\ 0 & y & x^3 & 0 & 0 \\ 0 & y & 0 & x^4 & 0 \\ 1 & y & 1 & 1 & 1 \end{bmatrix}$$

`Type: Matrix Polynomial Integer`

Use **copy** to make a copy of a matrix.

```
cdm := copy(dm)
```

$$\begin{bmatrix} 1 & y & 0 & 0 & 0 \\ 0 & y & 0 & 0 & 0 \\ 0 & y & x^3 & 0 & 0 \\ 0 & y & 0 & x^4 & 0 \\ 1 & y & 1 & 1 & 1 \end{bmatrix}$$

`Type: Matrix Polynomial Integer`

This is useful if you intend to modify a matrix destructively but want a copy of the original.

```
setelt(dm,4,1,1-x**7)
```

$$-x^7+1$$

Type: Polynomial Integer

```
[dm,cdm]
```

$$\left[\begin{bmatrix} 1 & y & 0 & 0 & 0 \\ 0 & y & 0 & 0 & 0 \\ 0 & y & x^3 & 0 & 0 \\ -x^7+1 & y & 0 & x^4 & 0 \\ 1 & y & 1 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & y & 0 & 0 & 0 \\ 0 & y & 0 & 0 & 0 \\ 0 & y & x^3 & 0 & 0 \\ 0 & y & 0 & x^4 & 0 \\ 1 & y & 1 & 1 & 1 \end{bmatrix}\right]$$

Type: List Matrix Polynomial Integer

Use **subMatrix** to extract part of an existing matrix. The syntax is `subMatrix(`*m, firstrow, lastrow, firstcol, lastcol*`)`.

```
subMatrix(dm,2,3,2,4)
```

$$\begin{bmatrix} y & 0 & 0 \\ y & x^3 & 0 \end{bmatrix}$$

Type: Matrix Polynomial Integer

To change a submatrix, use **setsubMatrix**.

```
d := diagonalMatrix [1.2,-1.3,1.4,-1.5]
```

$$\begin{bmatrix} 1.2 & 0.0 & 0.0 & 0.0 \\ 0.0 & -1.3 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.4 & 0.0 \\ 0.0 & 0.0 & 0.0 & -1.5 \end{bmatrix}$$

Type: Matrix Float

If `e` is too big to fit where you specify, an error message is displayed. Use **subMatrix** to extract part of `e`, if necessary.

```
e := matrix [ [6.7,9.11],[-31.33,67.19] ]
```

$$\begin{bmatrix} 6.7 & 9.11 \\ -31.33 & 67.19 \end{bmatrix}$$

Type: Matrix Float

This changes the submatrix of `d` whose upper left corner is at the first row and second column and whose size is that of `e`.

```
setsubMatrix!(d,1,2,e)
```

$$\begin{bmatrix} 1.2 & 6.7 & 9.11 & 0.0 \\ 0.0 & -31.33 & 67.19 & 0.0 \\ 0.0 & 0.0 & 1.4 & 0.0 \\ 0.0 & 0.0 & 0.0 & -1.5 \end{bmatrix}$$

Type: Matrix Float

```
d
```

$$\begin{bmatrix} 1.2 & 6.7 & 9.11 & 0.0 \\ 0.0 & -31.33 & 67.19 & 0.0 \\ 0.0 & 0.0 & 1.4 & 0.0 \\ 0.0 & 0.0 & 0.0 & -1.5 \end{bmatrix}$$

Type: Matrix Float

Matrices can be joined either horizontally or vertically to make new matrices.

```
a := matrix [ [1/2,1/3,1/4],[1/5,1/6,1/7] ]
```

$$\begin{bmatrix} \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{bmatrix}$$

Type: Matrix Fraction Integer

```
b := matrix [ [3/5,3/7,3/11],[3/13,3/17,3/19] ]
```

$$\begin{bmatrix} \frac{3}{5} & \frac{3}{7} & \frac{3}{11} \\ \frac{3}{13} & \frac{3}{17} & \frac{3}{19} \end{bmatrix}$$

Type: Matrix Fraction Integer

Use **horizConcat** to append them side to side. The two matrices must have the same number of rows.

```
horizConcat(a,b)
```

$$\begin{bmatrix} \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{3}{5} & \frac{3}{7} & \frac{3}{11} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{3}{13} & \frac{3}{17} & \frac{3}{19} \end{bmatrix}$$

Type: Matrix Fraction Integer

Use **vertConcat** to stack one upon the other. The two matrices must have the same number of columns.

```
vab := vertConcat(a,b)
```

$$\begin{bmatrix} \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{3}{5} & \frac{3}{7} & \frac{3}{11} \\ \frac{3}{13} & \frac{3}{17} & \frac{3}{19} \end{bmatrix}$$

Type: Matrix Fraction Integer

The operation **transpose** is used to create a new matrix by reflection across the main diagonal.

```
transpose vab
```

$$\begin{bmatrix} \frac{1}{2} & \frac{1}{5} & \frac{3}{5} & \frac{3}{13} \\ \frac{1}{3} & \frac{1}{6} & \frac{3}{7} & \frac{3}{17} \\ \frac{1}{4} & \frac{1}{7} & \frac{3}{11} & \frac{3}{19} \end{bmatrix}$$

Type: Matrix Fraction Integer

9.39.2 Operations on Matrices

AXIOM provides both left and right scalar multiplication.

```
m := matrix [ [1,2],[3,4] ]
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Type: Matrix Integer

```
4 * m * (-5)
```

$$\begin{bmatrix} -20 & -40 \\ -60 & -80 \end{bmatrix}$$

Type: Matrix Integer

You can add, subtract, and multiply matrices provided, of course, that the matrices have compatible dimensions. If not, an error message is displayed.

```
n := matrix([ [1,0,-2],[-3,5,1] ])
```

$$\begin{bmatrix} 1 & 0 & -2 \\ -3 & 5 & 1 \end{bmatrix}$$

Type: Matrix Integer

This following product is defined but `n * m` is not.

```
m * n
```

$$\begin{bmatrix} -5 & 10 & 0 \\ -9 & 20 & -2 \end{bmatrix}$$

Type: Matrix Integer

The operations **nrows** and **ncols** return the number of rows and columns of a matrix. You can extract a row or a column of a matrix using the operations **row** and **column**. The object returned is a `Vector`.

Here is the third column of the matrix `n`.

```
vec := column(n,3)
```

$$[-2, 1]$$

Type: Vector Integer

You can multiply a matrix on the left by a "row vector" and on the right by a "column vector."

```
vec * m
```

$$[1, 0]$$

Type: Vector Integer

Of course, the dimensions of the vector and the matrix must be compatible or an error message is returned.

```
m * vec
```

$$[0, -2]$$

Type: Vector Integer

The operation **inverse** computes the inverse of a matrix if the matrix is invertible, and returns `"failed"` if not.

This Hilbert matrix is invertible.

```
hilb := matrix([ [1/(i + j) for i in 1..3] for j in 1..3])
```

$$\begin{bmatrix} \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \end{bmatrix}$$

Type: Matrix Fraction Integer

```
inverse(hilb)
```

$$\begin{bmatrix} 72 & -240 & 180 \\ -240 & 900 & -720 \\ 180 & -720 & 600 \end{bmatrix}$$

Type: Union(Matrix Fraction Integer,...)

This matrix is not invertible.

```
mm := matrix([ [1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16]
])
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Type: Matrix Integer

```
inverse(mm)
```

"failed"

Type: Union("failed",...)

The operation **determinant** computes the determinant of a matrix provided that the entries of the matrix belong to a `CommutativeRing`.

The above matrix `mm` is not invertible and, hence, must have determinant `0`.

```
determinant(mm)
```

$$0$$

Type: NonNegativeInteger

The operation **trace** computes the trace of a *square* matrix.

```
trace(mm)
```

$$34$$

Type: PositiveInteger

The operation **rank** computes the *rank* of a matrix: the maximal number of linearly independent rows or columns.

```
rank(mm)
```

$$2$$

Type: PositiveInteger

The operation **nullity** computes the *nullity* of a matrix: the dimension of its null space.

```
nullity(mm)
```

$$2$$

Type: PositiveInteger

The operation **nullSpace** returns a list containing a basis for the null space of a matrix. Note that the nullity is the number of elements in a basis for the null space.

```
nullSpace(mm)
```

$$[[1, -2, 1, 0], [2, -3, 0, 1]]$$

Type: List Vector Integer

The operation **rowEchelon** returns the row echelon form of a matrix. It is easy to see that the rank of this matrix is two and that its nullity is also two.

```
rowEchelon(mm)
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 4 & 8 & 12 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Type: Matrix Integer

For more information on related topics, see 9.48 on page 173, 9.69 on page 233, 9.44 on page 158, and 9.66 on page 221. Issue the system command `)show Matrix` to display the full list of operations defined by `Matrix`.

9.40 MultiSet

The domain `Multiset(R)` is similar to `Set(R)` except that multiplicities (counts of duplications) are maintained and displayed. Use the operation **multiset** to create multisets from lists. All the standard operations from sets are available for multisets. An element with multiplicity greater than one has the multiplicity displayed first, then a colon, and then the element.

Create a multiset of integers.

```
s := multiset [1,2,3,4,5,4,3,2,3,4,5,6,7,4,10]
```

$$\{1, 2 : 2, 3 : 3, 4 : 4, 2 : 5, 6, 7, 10\}$$

Type: Multiset PositiveInteger

The operation `insert!` adds an element to a multiset.

```
insert!(3,s)
```

$$\{1, 2 : 2, 4 : 3, 4 : 4, 2 : 5, 6, 7, 10\}$$

Type: Multiset PositiveInteger

Use `remove!` to remove an element. If a third argument is present, it specifies how many instances to remove. Otherwise all instances of the element are removed. Display the resulting multiset.

```
remove!(5,s); s
```

$$\{1, 2 : 2, 3 : 3, 4 : 4, 5, 6, 7, 10\}$$

Type: Multiset PositiveInteger

The operation `count` returns the number of copies of a given value.

```
count(5,s)
```

$$1$$

Type: NonNegativeInteger

A second multiset.

```
t := multiset [2,2,2,-9]
```

$$\{3\colon\ 2, -9\}$$

Type: Multiset Integer

The `union` of two multisets is additive.

```
U := union(s,t)
```

$$\{1, 5 : 2, 4 : 3, 4 : 4, 5, 6, 7, 10, -9\}$$

Type: Multiset Integer

The `intersect` operation gives the elements that are in common, with additive multiplicity.

```
I := intersect(s,t)
```

$$\{5\colon 2\}$$

Type: Multiset Integer

The `difference` of `s` and `t` consists of the elements that `s` has but `t` does not. Elements are regarded as indistinguishable, so that if `s` and `t` have any element in common, the `difference` does not contain that element.

```
difference(s,t)
```

$$\{1, 4 : 3, 4 : 4, 5, 6, 7, 10\}$$

Type: Multiset Integer

The `symmetricDifference` is the `union` of `difference(s, t)` and `difference(t, s)`.

```
S := symmetricDifference(s,t)
```

$$\{1, 4 : 3, 4 : 4, 5, 6, 7, 10, -9\}$$

Type: Multiset Integer

Check that the `union` of the `symmetricDifference` and the `intersect` equals the `union` of the elements.

```
(U = union(S,I))@Boolean
```

true

Type: Boolean

Check some inclusion relations.

```
t1 := multiset [1,2,2,3]; [t1 < t, t1 < s, t < s, t1 <= s]
```

[false, true, false, true]

Type: List Boolean

9.41 MultivariatePolynomial

The domain constructor MultivariatePolynomial is similar to Polynomial except that it specifies the variables to be used. Polynomial are available for MultivariatePolynomial. The abbreviation for MultivariatePolynomial is MPOLY. The type expressions MultivariatePolynomial([x,y],Integer) and

MPOLY([x,y],INT)

refer to the domain of multivariate polynomials in the variables x and y where the coefficients are restricted to be integers. The first variable specified is the main variable and the display of the polynomial reflects this.

This polynomial appears with terms in descending powers of the variable x.

```
m :  MPOLY([x,y],INT) := (x**2 - x*y**3 +3*y)**2
```

$$x^4 - 2\ y^3\ x^3 + \left(y^6 + 6\ y\right)\ x^2 - 6\ y^4\ x + 9\ y^2$$

Type: MultivariatePolynomial([x,y],Integer)

It is easy to see a different variable ordering by doing a conversion.

```
m ::  MPOLY([y,x],INT)
```

$$x^2\ y^6 - 6\ x\ y^4 - 2\ x^3\ y^3 + 9\ y^2 + 6\ x^2\ y + x^4$$

Type: MultivariatePolynomial([y,x],Integer)

You can use other, unspecified variables, by using Polynomial in the coefficient type of MPOLY.

```
p :  MPOLY([x,y],POLY INT)
```

Type: Void

```
p := (a**2*x - b*y**2 + 1)**2
```

$$a^4\ x^2 + \left(-2\ a^2\ b\ y^2 + 2\ a^2\right)\ x + b^2\ y^4 - 2\ b\ y^2 + 1$$

Type: MultivariatePolynomial([x,y],Polynomial Integer)

Conversions can be used to re-express such polynomials in terms of the other variables. For example, you can first push all the variables into a polynomial with integer coefficients.

```
p ::  POLY INT
```

$$b^2\ y^4 + \left(-2\ a^2\ b\ x - 2\ b\right)\ y^2 + a^4\ x^2 + 2\ a^2\ x + 1$$

Type: Polynomial Integer

Now pull out the variables of interest.

```
% :: MPOLY([a,b],POLY INT)
```

$$x^2\ a^4 + \left(-2\ x\ y^2\ b + 2\ x\right)\ a^2 + y^4\ b^2 - 2\ y^2\ b + 1$$

Type: MultivariatePolynomial([a,b],Polynomial Integer)

Restriction:

> AXIOM does not allow you to create types where `Multivariate Polynomial` is contained in the coefficient type of `Polynomial`. Therefore, `MPOLY([x,y],POLY INT)` is legal but `POLY MPOLY ([x,y],INT)` is not.

.

Multivariate polynomials may be combined with univariate polynomials to create types with special structures.

```
q : UP(x, FRAC MPOLY([y,z],INT))
```

Void

This is a polynomial in `x` whose coefficients are quotients of polynomials in `y` and `z`.

```
q := (x**2 - x*(z+1)/y +2)**2
```

$$x^4 + \frac{-2\ z - 2}{y}\ x^3 + \frac{4\ y^2 + z^2 + 2\ z + 1}{y^2}\ x^2 + \frac{-4\ z - 4}{y}\ x + 4$$

Type: UnivariatePolynomial(x,Fraction MultivariatePolynomial([y,z],Integer))

Use conversions for structural rearrangements. `z` does not appear in a denominator and so it can be made the main variable.

```
q :: UP(z, FRAC MPOLY([x,y],INT))
```

$$\frac{x^2}{y^2}\ z^2 + \frac{-2\ y\ x^3 + 2\ x^2 - 4\ y\ x}{y^2}\ z +$$

$$\frac{y^2\ x^4 - 2\ y\ x^3 + \left(4\ y^2 + 1\right)\ x^2 - 4\ y\ x + 4\ y^2}{y^2}$$

Type: UnivariatePolynomial(z,Fraction MultivariatePolynomial([x,y],Integer))

Or you can make a multivariate polynomial in x and z whose coefficients are fractions in polynomials in y.

```
q :: MPOLY([x,z], FRAC UP(y,INT))
```

$$x^4+\left(-\frac{2}{y}\ z-\frac{2}{y}\right)\ x^3+\left(\frac{1}{y^2}\ z^2+\frac{2}{y^2}\ z+\frac{4\ y^2+1}{y^2}\right)\ x^2+$$

$$\left(-\frac{4}{y}\ z-\frac{4}{y}\right)\ x+4$$

Type: MultivariatePolynomial([x,z],Fraction UnivariatePolynomial(y,Integer))

A conversion like `q :: MPOLY([x,y], FRAC UP(z,INT))` is not possible in this example because y appears in the denominator of a fraction. As you can see, AXIOM provides extraordinary flexibility in the manipulation and display of expressions via its conversion facility.

For more information on related topics, see 9.49 on page 174, 9.67 on page 225, and 9.15 on page 59.

9.42 None

The `None` domain is not very useful for interactive work but it is provided nevertheless for completeness of the AXIOM type system.

Probably the only place you will ever see it is if you enter an empty list with no type information.

```
[ ]
```

$$[\,]$$

Type: List None

Such an empty list can be converted into an empty list of any other type.

```
[ ] :: List Float
```

$$[\,]$$

Type: List Float

If you wish to produce an empty list of a particular type directly, such as `List NonNegativeInteger`, do it this way.

```
[ ]$List(NonNegativeInteger)
```

$$[\,]$$

Type: List NonNegativeInteger

9.43 Octonion

The Octonions, also called the Cayley-Dixon algebra, defined over a commutative ring are an eight-dimensional non-associative algebra. Their construction from quaternions is similar to the construction of quaternions from complex numbers (see 9.50 on page 182).

As `Octonion` creates an eight-dimensional algebra, you have to give eight components to construct an octonion.

```
oci1 := octon(1,2,3,4,5,6,7,8)
```

$$1 + 2\ i + 3\ j + 4\ k + 5\ E + 6\ I + 7\ J + 8\ K$$

Type: Octonion Integer

```
oci2 := octon(7,2,3,-4,5,6,-7,0)
```

$$7 + 2\ i + 3\ j - 4\ k + 5\ E + 6\ I - 7\ J$$

Type: Octonion Integer

Or you can use two quaternions to create an octonion.

```
oci3 := octon(quatern(-7,-12,3,-10), quatern(5,6,9,0))
```

$$-7 - 12\ i + 3\ j - 10\ k + 5\ E + 6\ I + 9\ J$$

Type: Octonion Integer

You can easily demonstrate the non-associativity of multiplication.

```
(oci1 * oci2) * oci3 - oci1 * (oci2 * oci3)
```

$$2696\ i - 2928\ j - 4072\ k + 16\ E - 1192\ I + 832\ J + 2616\ K$$

Type: Octonion Integer

As with the quaternions, we have a real part, the imaginary parts `i`, `j`, `k`, and four additional imaginary parts `E`, `I`, `J` and `K`. These parts correspond to the canonical basis `(1,i,j,k,E,I,J,K)`.

For each basis element there is a component operation to extract the coefficient of the basis element for a given octonion.

```
[real oci1, imagi oci1, imagj oci1, imagk oci1, imagE oci1, imagI
oci1, imagJ oci1, imagK oci1]
```

$$[1, 2, 3, 4, 5, 6, 7, 8]$$

Type: List PositiveInteger

A basis with respect to the quaternions is given by (1,E). However, you might ask, what then are the commuting rules? To answer this, we create some generic elements.

We do this in AXIOM by simply changing the ground ring from `Integer` to `Polynomial Integer`.

```
q :  Quaternion Polynomial Integer := quatern(q1, qi, qj, qk)
```

$$q1 + qi\ i + qj\ j + qk\ k$$

Type: Quaternion Polynomial Integer

```
E : Octonion Polynomial Integer:= octon(0,0,0,0,1,0,0,0)
```

$$E$$

Type: Octonion Polynomial Integer

Note that quaternions are automatically converted to octonions in the obvious way.

```
q * E
```

$$q1\ E + qi\ I + qj\ J + qk\ K$$

Type: Octonion Polynomial Integer

```
E * q
```

$$q1\ E - qi\ I - qj\ J - qk\ K$$

Type: Octonion Polynomial Integer

```
q * 1$(Octonion Polynomial Integer)
```

$$q1 + qi\ i + qj\ j + qk\ k$$

Type: Octonion Polynomial Integer

```
1$(Octonion Polynomial Integer) * q
```

$$q1 + qi\ i + qj\ j + qk\ k$$

Type: Octonion Polynomial Integer

Finally, we check that the **norm**, defined as the sum of the squares of the coefficients, is a multiplicative map.

```
o : Octonion Polynomial Integer := octon(o1, oi, oj, ok, oE, oI,
oJ, oK)
```

$$o1 + oi\ i + oj\ j + ok\ k + oE\ E + oI\ I + oJ\ J + oK\ K$$

Type: Octonion Polynomial Integer

```
norm o
```

$$ok^2 + oj^2 + oi^2 + oK^2 + oJ^2 + oI^2 + oE^2 + o1^2$$

Type: Polynomial Integer

```
p : Octonion Polynomial Integer := octon(p1, pi, pj, pk, pE, pI,
pJ, pK)
```

$$p1 + pi\ i + pj\ j + pk\ k + pE\ E + pI\ I + pJ\ J + pK\ K$$

Type: Octonion Polynomial Integer

Since the result is `0`, the norm is multiplicative.

```
norm(o*p)-norm(p)*norm(o)
```

$$0$$

Type: Polynomial Integer

9.44 OneDimensionalArray

The `OneDimensionalArray` domain is used for storing data in a one-dimensional indexed data structure. Such an array is a homogeneous data structure in that all the entries of the array must belong to the same AXIOM domain. Each array has a fixed length specified by the user and arrays are not extensible. The indexing of one-dimensional arrays is one-based. This means that the "first" element of an array is given the index `1`. See also 9.69 on page 233 and 9.23 on page 79.

To create a one-dimensional array, apply the operation `oneDimensional Array` to a list.

```
oneDimensionalArray [i**2 for i in 1..10]
```

$$[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]$$

Type: OneDimensionalArray PositiveInteger

Another approach is to first create `a`, a one-dimensional array of 10 `0`'s. `OneDimensionalArray` has the convenient abbreviation `ARRAY1`.

```
a :  ARRAY1 INT := new(10,0)
```

$$[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

Type: `OneDimensionalArray Integer`

Set each `i`th element to i, then display the result.

```
for i in 1..10 repeat a.i := i; a
```

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Type: `OneDimensionalArray Integer`

Square each element by mapping the function $i \mapsto i^2$ onto each element.

```
map!(i +-> i ** 2,a); a
```

$$[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]$$

Type: `OneDimensionalArray Integer`

Reverse the elements in place.

```
reverse!  a
```

$$[100, 81, 64, 49, 36, 25, 16, 9, 4, 1]$$

Type: `OneDimensionalArray Integer`

Swap the `4`th and `5`th element.

```
swap!(a,4,5); a
```

$$[100, 81, 64, 36, 49, 25, 16, 9, 4, 1]$$

Type: `OneDimensionalArray Integer`

Sort the elements in place.

```
sort!  a
```

$$[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]$$

Type: `OneDimensionalArray Integer`

Create a new one-dimensional array `b` containing the last 5 elements of `a`.

```
b := a(6..10)
```

$$[36, 49, 64, 81, 100]$$

Type: OneDimensionalArray Integer

Replace the first 5 elements of `a` with those of `b`.

```
copyInto!(a,b,1)
```

$$[36, 49, 64, 81, 100, 36, 49, 64, 81, 100]$$

Type: OneDimensionalArray Integer

9.45 Operator

Given any ring `R`, the ring of the `Integer`-linear operators over `R` is called `Operator(R)`. To create an operator over `R`, first create a basic operator using the operation `operator`, and then convert it to `Operator(R)` for the `R` you want.

We choose `R` to be the two by two matrices over the integers.

```
R := SQMATRIX(2, INT)
```

$$SquareMatrix(2, Integer)$$

Type: Domain

Create the operator `tilde` on `R`.

```
t := operator("tilde") :: OP(R)
```

$$tilde$$

Type: Operator SquareMatrix(2,Integer)

To attach an evaluation function (from `R` to `R`) to an operator over `R`, use `evaluate(op, f)` where `op` is an operator over `R` and `f` is a function `R -> R`. This needs to be done only once when the operator is defined. Note that `f` must be `Integer`-linear (that is, `f(ax+y) = a f(x) + f(y)` for any integer `a`, and any `x` and `y` in `R`).

We now attach the transpose map to the above operator `t`.

```
evaluate(t, m +-> transpose m)
```

$$tilde$$

Type: Operator SquareMatrix(2,Integer)

Operators can be manipulated formally as in any ring: + is the pointwise addition and * is composition. Any element x of R can be converted to an operator op_x over R, and the evaluation function of op_x is left-multiplication by x.

Multiplying on the left by this matrix swaps the two rows.

```
s :  R := matrix [ [0, 1], [1, 0] ]
```

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Type: SquareMatrix(2,Integer)

Can you guess what is the action of the following operator?

```
rho := t * s
```

$$tilde \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Type: Operator SquareMatrix(2,Integer)

Hint: applying rho four times gives the identity, so rho**4-1 should return 0 when applied to any two by two matrix.

```
z := rho**4 - 1
```

$$-1 + tilde \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} tilde \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} tilde \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} tilde \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Type: Operator SquareMatrix(2,Integer)

Now check with this matrix.

```
m:R := matrix [ [1, 2], [3, 4] ]
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Type: SquareMatrix(2,Integer)

```
z m
```

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Type: SquareMatrix(2,Integer)

As you have probably guessed by now, rho acts on matrices by rotating the elements clockwise.

```
rho m
```

$$\begin{bmatrix} 3 & 1 \\ 4 & 2 \end{bmatrix}$$

Type: SquareMatrix(2,Integer)

```
rho rho m
```

$$\begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix}$$

Type: SquareMatrix(2,Integer)

```
(rho**3) m
```

$$\begin{bmatrix} 2 & 4 \\ 1 & 3 \end{bmatrix}$$

Type: SquareMatrix(2,Integer)

Do the swapping of rows and transposition commute? We can check by computing their bracket.

```
b := t * s - s * t
```

$$-\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} tilde + tilde \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Type: Operator SquareMatrix(2,Integer)

Now apply it to `m`.

```
b m
```

$$\begin{bmatrix} 1 & -3 \\ 3 & -1 \end{bmatrix}$$

Type: SquareMatrix(2,Integer)

Next we demonstrate how to define a differential operator on a polynomial ring.

This is the recursive definition of the `n`-th Legendre polynomial.

```
L n ==
  n = 0 => 1
  n = 1 => x
  (2*n-1)/n * x * L(n-1) - (n-1)/n * L(n-2)
```

Type: Void

Create the differential operator $\frac{d}{dx}$ on polynomials in `x` over the rational numbers.

```
dx := operator("D") :: OP(POLY FRAC INT)
```

$$D$$

Type: Operator Polynomial Fraction Integer

Now attach the map to it.

```
evaluate(dx, p +-> D(p, 'x))
```

$$D$$

Type: Operator Polynomial Fraction Integer

This is the differential equation satisfied by the `n`-th Legendre polynomial.

```
E n == (1 - x**2) * dx**2 - 2 * x * dx + n*(n+1)
```

Void

Now we verify this for `n = 15`. Here is the polynomial.

```
L 15
```

$$\frac{9694845}{2048}\,x^{15} - \frac{35102025}{2048}\,x^{13} + \frac{50702925}{2048}\,x^{11} - \frac{37182145}{2048}\,x^{9} + \frac{14549535}{2048}\,x^{7} - \frac{2909907}{2048}\,x^{5} + \frac{255255}{2048}\,x^{3} - \frac{6435}{2048}\,x$$

Type: Polynomial Fraction Integer

Here is the operator.

```
E 15
```

$$240 - 2\,x\,D + \left(-x^2 + 1\right)\,D^2$$

Type: Operator Polynomial Fraction Integer

Here is the evaluation.

```
(E 15)(L 15)
```

$$0$$

Type: Polynomial Fraction Integer

9.46 OrderlyDifferentialPolynomial

Many systems of differential equations may be transformed to equivalent systems of ordinary differential equations where the equations are expressed polynomially in terms of the unknown functions. In AXIOM, the domain constructors `OrderlyDifferentialPolynomial` (abbreviated `ODPOL`) and `SequentialDifferentialPolynomial` (abbreviation `SDPOL`) implement two domains of ordinary differential polynomials over any differential ring. In the simplest case, this differential ring is usually either the ring of integers, or the field of rational numbers. However, AXIOM can handle ordinary differential polynomials over a field of rational functions in a single indeterminate.

The two domains `ODPOL` and `SDPOL` are almost identical, the only difference being the choice of a different ranking, which is an ordering of the derivatives of the indeterminates. The first domain uses an orderly ranking, that is, derivatives of higher order are ranked higher, and derivatives of the same order are ranked alphabetically. The second domain uses a sequential ranking, where derivatives are ordered first alphabetically by the differential indeterminates, and then by order. A more general domain constructor, `DifferentialSparseMultivariatePolynomial` (abbreviation `DSMP`) allows both a user-provided list of differential indeterminates as well as a user-defined ranking. We shall illustrate `ODPOL(FRAC INT)`, which constructs a domain of ordinary differential polynomials in an arbitrary number of differential indeterminates with rational numbers as coefficients.

```
dpol:= ODPOL(FRAC INT)
```

OrderlyDifferentialPolynomial Fraction Integer

```
Type: Domain
```

A differential indeterminate `w` may be viewed as an infinite sequence of algebraic indeterminates, which are the derivatives of `w`. To facilitate referencing these, AXIOM provides the operation **makeVariable** to convert an element of type `Symbol` to a map from the natural numbers to the differential polynomial ring.

```
w := makeVariable('w)$dpol
```

theMap(...)

```
Type: (NonNegativeInteger -> OrderlyDifferentialPolynomial
                                          Fraction Integer)
```

```
z := makeVariable('z)$dpol
```

theMap(...)

```
Type: (NonNegativeInteger -> OrderlyDifferentialPolynomial
                                          Fraction Integer)
```

The fifth derivative of `w` can be obtained by applying the map `w` to the number `5`. Note that the order of differentiation is given as a subscript (except when the order is 0).

```
w.5
```

$$w_5$$

Type: `OrderlyDifferentialPolynomial Fraction Integer`

```
w 0
```

$$w$$

Type: `OrderlyDifferentialPolynomial Fraction Integer`

The first five derivatives of `z` can be generated by a list.

```
[z.i for i in 1..5]
```

$$[z_1, z_2, z_3, z_4, z_5]$$

Type: `List OrderlyDifferentialPolynomial Fraction Integer`

The usual arithmetic can be used to form a differential polynomial from the derivatives.

```
f:= w.4 - w.1 * w.1 * z.3
```

$$w_4 - {w_1}^2 \ z_3$$

Type: `OrderlyDifferentialPolynomial Fraction Integer`

```
g:=(z.1)**3 * (z.2)**2 - w.2
```

$${z_1}^3 \ {z_2}^2 - w_2$$

Type: `OrderlyDifferentialPolynomial Fraction Integer`

The operation **D** computes the derivative of any differential polynomial.

```
D(f)
```

$$w_5 - {w_1}^2 \ z_4 - 2 \ w_1 \ w_2 \ z_3$$

Type: `OrderlyDifferentialPolynomial Fraction Integer`

The same operation can compute higher derivatives, like the fourth derivative.

```
D(f,4)
```

$$w_8 - {w_1}^2\ z_7 - 8\ w_1\ w_2\ z_6 + \left(-12\ w_1\ w_3 - 12\ {w_2}^2\right)\ z_5 - 2\ w_1\ z_3\ w_5 +$$
$$(-8\ w_1\ w_4 - 24\ w_2\ w_3)\ z_4 - 8\ w_2\ z_3\ w_4 - 6\ {w_3}^2\ z_3$$

Type: OrderlyDifferentialPolynomial Fraction Integer

The operation **makeVariable** creates a map to facilitate referencing the derivatives of `f`, similar to the map `w`.

```
df:=makeVariable(f)$dpol
```

theMap(...)

Type: (NonNegativeInteger -> OrderlyDifferentialPolynomial Fraction Integer)

The fourth derivative of f may be referenced easily.

```
df.4
```

$$w_8 - {w_1}^2\ z_7 - 8\ w_1\ w_2\ z_6 + \left(-12\ w_1\ w_3 - 12\ {w_2}^2\right)\ z_5 - 2\ w_1\ z_3\ w_5 +$$
$$(-8\ w_1\ w_4 - 24\ w_2\ w_3)\ z_4 - 8\ w_2\ z_3\ w_4 - 6\ {w_3}^2\ z_3$$

Type: OrderlyDifferentialPolynomial Fraction Integer

The operation **order** returns the order of a differential polynomial, or the order in a specified differential indeterminate.

```
order(g)
```

$$2$$

Type: PositiveInteger

```
order(g, 'w)
```

$$2$$

Type: PositiveInteger

The operation **differentialVariables** returns a list of differential indeterminates occurring in a differential polynomial.

```
differentialVariables(g)
```

$$[z, w]$$

Type: List Symbol

The operation **degree** returns the degree, or the degree in the differential indeterminate specified.

```
degree(g)
```

$$ {z_2}^2 \ {z_1}^3 $$

Type: IndexedExponents OrderlyDifferentialVariable Symbol

```
degree(g, 'w)
```

$$ 1 $$

Type: PositiveInteger

The operation **weights** returns a list of weights of differential monomials appearing in differential polynomial, or a list of weights in a specified differential indeterminate.

```
weights(g)
```

$$ [7, 2] $$

Type: List NonNegativeInteger

```
weights(g,'w)
```

$$ [2] $$

Type: List NonNegativeInteger

The operation **weight** returns the maximum weight of all differential monomials appearing in the differential polynomial.

```
weight(g)
```

$$ 7 $$

Type: PositiveInteger

A differential polynomial is *isobaric* if the weights of all differential monomials appearing in it are equal.

```
isobaric?(g)
```

false

Type: Boolean

To substitute *differentially*, use **eval**. Note that we must coerce `'w` to `Symbol`, since in `ODPOL`, differential indeterminates belong to the domain `Symbol`. Compare this result to the next, which substitutes *algebraically* (no substitution is done since `w.0` does not appear in `g`).

```
eval(g,['w::Symbol],[f])
```

$$-w_6 + {w_1}^2\ z_5 + 4\ w_1\ w_2\ z_4 + \left(2\ w_1\ w_3 + 2\ {w_2}^2\right)\ z_3 + {z_1}^3\ {z_2}^2$$

Type: OrderlyDifferentialPolynomial Fraction Integer

```
eval(g,['w],[f])
```

$${z_1}^3\ {z_2}^2 - w_2$$

Type: OrderlyDifferentialPolynomial Fraction Integer

Since `OrderlyDifferentialPolynomial` belongs to `PolynomialCategory`, all the operations defined in the latter category, or in packages for the latter category, are available.

```
monomials(g)
```

$$\left[{z_1}^3\ {z_2}^2, -w_2\right]$$

Type: List OrderlyDifferentialPolynomial Fraction Integer

```
variables(g)
```

$$[z_2, w_2, z_1]$$

Type: List OrderlyDifferentialVariable Symbol

```
gcd(f,g)
```

$$1$$

Type: OrderlyDifferentialPolynomial Fraction Integer

```
groebner([f,g])
```

$$\left[w_4 - {w_1}^2\ z_3, {z_1}^3\ {z_2}^2 - w_2\right]$$

Type: List OrderlyDifferentialPolynomial Fraction Integer

The next three operations are essential for elimination procedures in differential polynomial rings. The operation **leader** returns the leader of a differential polynomial, which is the highest ranked derivative of the differential indeterminates that occurs.

```
lg:=leader(g)
```

$$z_2$$

Type: OrderlyDifferentialVariable Symbol

The operation **separant** returns the separant of a differential polynomial, which is the partial derivative with respect to the leader.

```
sg:=separant(g)
```

$$2\ {z_1}^3\ z_2$$

Type: OrderlyDifferentialPolynomial Fraction Integer

The operation **initial** returns the initial, which is the leading coefficient when the given differential polynomial is expressed as a polynomial in the leader.

```
ig:=initial(g)
```

$${z_1}^3$$

Type: OrderlyDifferentialPolynomial Fraction Integer

Using these three operations, it is possible to reduce `f` modulo the differential ideal generated by `g`. The general scheme is to first reduce the order, then reduce the degree in the leader. First, eliminate `z.3` using the derivative of `g`.

```
g1 := D g
```

$$2\ {z_1}^3\ z_2\ z_3 - w_3 + 3\ {z_1}^2\ {z_2}^3$$

Type: OrderlyDifferentialPolynomial Fraction Integer

Find its leader.

```
lg1:= leader g1
```

$$z_3$$

Type: OrderlyDifferentialVariable Symbol

Differentiate `f` partially with respect to this leader.

```
pdf:=D(f, lg1)
```

$$-{w_1}^2$$

Type: OrderlyDifferentialPolynomial Fraction Integer

Compute the partial remainder of f with respect to g.

```
prf:=sg * f- pdf * g1
```

$$2\ {z_1}^3\ z_2\ w_4 - {w_1}^2\ w_3 + 3\ {w_1}^2\ {z_1}^2\ {z_2}^3$$

Type: OrderlyDifferentialPolynomial Fraction Integer

Note that high powers of lg still appear in prf. Compute the leading coefficient of prf as a polynomial in the leader of g.

```
lcf:=leadingCoefficient univariate(prf, lg)
```

$$3\ {w_1}^2\ {z_1}^2$$

Type: OrderlyDifferentialPolynomial Fraction Integer

Finally, continue eliminating the high powers of lg appearing in prf to obtain the (pseudo) remainder of f modulo g and its derivatives.

```
ig * prf - lcf * g * lg
```

$$2\ {z_1}^6\ z_2\ w_4 - {w_1}^2\ {z_1}^3\ w_3 + 3\ {w_1}^2\ {z_1}^2\ w_2\ z_2$$

Type: OrderlyDifferentialPolynomial Fraction Integer

9.47 PartialFraction

A *partial fraction* is a decomposition of a quotient into a sum of quotients where the denominators of the summands are powers of primes.[5] For example, the rational number 1/6 is decomposed into 1/2 -1/3. You can compute partial fractions of quotients of objects from domains belonging to the category Euclidean Domain. For example, Integer, Complex Integer, and UnivariatePolynomial (x, Fraction Integer) all belong to EuclideanDomain. In the examples following, we demonstrate how to decompose quotients of each of these kinds of object into partial fractions. Issue the system command)show PartialFraction to display the full list of operations defined by PartialFraction.

It is necessary that we know how to factor the denominator when we want to compute a partial fraction. Although the interpreter can often do this automatically, it may be necessary for you to include a call to factor. In these examples, it is not necessary to factor the denominators explicitly.

[5]Most people first encounter partial fractions when they are learning integral calculus. For a technical discussion of partial fractions, see, for example, Lang's *Algebra*.

The main operation for computing partial fractions is called **partialFraction** and we use this to compute a decomposition of `1 / 10!`. The first argument to **partialFraction** is the numerator of the quotient and the second argument is the factored denominator.

```
partialFraction(1,factorial 10)
```

$$\frac{159}{2^8} - \frac{23}{3^4} - \frac{12}{5^2} + \frac{1}{7}$$

Type: PartialFraction Integer

Since the denominators are powers of primes, it may be possible to expand the numerators further with respect to those primes. Use the operation **padicFraction** to do this.

```
f := padicFraction(%)
```

$$\frac{1}{2} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^6} + \frac{1}{2^7} + \frac{1}{2^8} - \frac{2}{3^2} - \frac{1}{3^3} - \frac{2}{3^4} - \frac{2}{5} - \frac{2}{5^2} + \frac{1}{7}$$

Type: PartialFraction Integer

The operation **compactFraction** returns an expanded fraction into the usual form. The compacted version is used internally for computational efficiency.

```
compactFraction(f)
```

$$\frac{159}{2^8} - \frac{23}{3^4} - \frac{12}{5^2} + \frac{1}{7}$$

Type: PartialFraction Integer

You can add, subtract, multiply and divide partial fractions. In addition, you can extract the parts of the decomposition. **numberOfFractionalTerms** computes the number of terms in the fractional part. This does not include the whole part of the fraction, which you get by calling **wholePart**. In this example, the whole part is just `0`.

```
numberOfFractionalTerms(f)
```

$$12$$

Type: PositiveInteger

The operation **nthFractionalTerm** returns the individual terms in the decomposition. Notice that the object returned is a partial fraction itself. **firstNumer** and **firstDenom** extract the numerator and denominator of the first term of the fraction.

```
nthFractionalTerm(f,3)
```

$$\frac{1}{2^5}$$

Type: PartialFraction Integer

Given two gaussian integers (see 9.10 on page 34), you can decompose their quotient into a partial fraction.

```
partialFraction(1,- 13 + 14 * %i)
```

$$-\frac{1}{1+2\ \%i}+\frac{4}{3+8\ \%i}$$

Type: PartialFraction Complex Integer

To convert back to a quotient, simply use a conversion.

```
% :: Fraction Complex Integer
```

$$-\frac{\%i}{14+13\ \%i}$$

Type: Fraction Complex Integer

To conclude this section, we compute the decomposition of

$$\frac{1}{(x+1)(x+2)^2(x+3)^3(x+4)^4}$$

The polynomials in this object have type `UnivariatePolynomial(x, Fraction Integer)`.

We use the **primeFactor** operation (see 9.19 on page 66) to create the denominator in factored form directly.

```
u : FR UP(x, FRAC INT) := reduce(*,[primeFactor(x+i,i) for i in
1..4])
```

$$(x+1)\ (x+2)^2\ (x+3)^3\ (x+4)^4$$

Type: Factored UnivariatePolynomial(x,Fraction Integer)

These are the compact and expanded partial fractions for the quotient.

```
partialFraction(1,u)
```

$$\frac{\frac{1}{648}}{x+1}+\frac{\frac{1}{4}\ x+\frac{7}{16}}{(x+2)^2}+\frac{-\frac{17}{8}\ x^2-12\ x-\frac{139}{8}}{(x+3)^3}+$$

$$\frac{\frac{607}{324}\ x^3+\frac{10115}{432}\ x^2+\frac{391}{4}\ x+\frac{44179}{324}}{(x+4)^4}$$

```
Type: PartialFraction UnivariatePolynomial(x,Fraction Integer)
```

```
padicFraction %
```

$$\frac{\frac{1}{648}}{x+1}+\frac{\frac{1}{4}}{x+2}-\frac{\frac{1}{16}}{(x+2)^2}-\frac{\frac{17}{8}}{x+3}+\frac{\frac{3}{4}}{(x+3)^2}-\frac{\frac{1}{2}}{(x+3)^3}+\frac{\frac{607}{324}}{x+4}+$$

$$\frac{\frac{403}{432}}{(x+4)^2}+\frac{\frac{13}{36}}{(x+4)^3}+\frac{\frac{1}{12}}{(x+4)^4}$$

```
Type: PartialFraction UnivariatePolynomial(x,Fraction Integer)
```

9.48 Permanent

The package `Permanent` provides the function **permanent** for square matrices. The **permanent** of a square matrix can be computed in the same way as the determinant by expansion of minors except that for the permanent the sign for each element is `1`, rather than being `1` if the row plus column indices is positive and `-1` otherwise. This function is much more difficult to compute efficiently than the **determinant**. An example of the use of **permanent** is the calculation of the n-th derangement number, defined to be the number of different possibilities for `n` couples to dance but never with their own spouse.

Consider an `n` by `n` matrix with entries `0` on the diagonal and `1` elsewhere. Think of the rows as one-half of each couple (for example, the males) and the columns the other half. The permanent of such a matrix gives the desired derangement number.

```
kn n ==
  r : MATRIX INT := new(n,n,1)
  for i in 1..n repeat
    r.i.i := 0
  r
```

```
Type: Void
```

Here are some derangement numbers, which you see grow quite fast.

```
permanent(kn(5) :: SQMATRIX(5,INT))
```

```
Compiling function kn with type PositiveInteger -> Matrix Integer
```

$$44$$

```
Type: PositiveInteger
```

```
[permanent(kn(n) :: SQMATRIX(n,INT)) for n in 1..13]
```

$$[0,1,2,9,44,265,1854,14833,133496,\\ 1334961,14684570,176214841,2290792932]$$

```
Type: List NonNegativeInteger
```

9.49 Polynomial

The domain constructor `Polynomial` (abbreviation: `POLY`) provides polynomials with an arbitrary number of unspecified variables.

It is used to create the default polynomial domains in AXIOM. Here the coefficients are integers.

```
x + 1
```

$$x + 1$$

Type: `Polynomial Integer`

Here the coefficients have type `Float`.

```
z - 2.3
```

$$z - 2.3$$

Type: `Polynomial Float`

And here we have a polynomial in two variables with coefficients which have type `Fraction Integer`.

```
y**2 - z + 3/4
```

$$-z + y^2 + \frac{3}{4}$$

Type: `Polynomial Fraction Integer`

The representation of objects of domains created by `Polynomial` is that of recursive univariate polynomials.[6]

This recursive structure is sometimes obvious from the display of a polynomial.

```
y **2 + x*y + y
```

$$y^2 + (x + 1)\ y$$

Type: `Polynomial Integer`

In this example, you see that the polynomial is stored as a polynomial in `y` with coefficients that are polynomials in `x` with integer coefficients. In fact, you really don't need to worry about the representation unless you are working on an advanced application where it is critical. The polynomial types created from `DistributedMultivariatePolynomial` and `NewDistributedMultivariatePolynomial` (discussed in 9.15 on page 59) are stored and displayed in a non-recursive manner.

You see a "flat" display of the above polynomial by converting to one of those types.

[6] The term `univariate` means "one variable." `multivariate` means "possibly more than one variable."

```
% :: DMP([y,x],INT)
```

$$y^2 + y\ x + y$$

Type: DistributedMultivariatePolynomial([y,x],Integer)

We will demonstrate many of the polynomial facilities by using two polynomials with integer coefficients.

By default, the interpreter expands polynomial expressions, even if they are written in a factored format.

```
p := (y-1)**2 * x * z
```

$$\left(x\ y^2 - 2\ x\ y + x\right)\ z$$

Type: Polynomial Integer

See 'Factored' on page 66 to see how to create objects in factored form directly.

```
q := (y-1) * x * (z+5)
```

$$(x\ y - x)\ z + 5\ x\ y - 5\ x$$

Type: Polynomial Integer

The fully factored form can be recovered by using **factor**.

```
factor(q)
```

$$x\ (y-1)\ (z+5)$$

Type: Factored Polynomial Integer

This is the same name used for the operation to factor integers. Such reuse of names is called and makes it much easier to think of solving problems in general ways. AXIOM facilities for factoring polynomials created with `Polynomial` are currently restricted to the integer and rational number coefficient cases.

The standard arithmetic operations are available for polynomials.

```
p - q**2
```

$$\left(-x^2\ y^2 + 2\ x^2\ y - x^2\right)\ z^2 +$$

$$\left(\left(-10\ x^2 + x\right)\ y^2 + \left(20\ x^2 - 2\ x\right)\ y - 10\ x^2 + x\right)\ z -$$

$$25\ x^2\ y^2 + 50\ x^2\ y - 25\ x^2$$

Type: Polynomial Integer

The operation **gcd** is used to compute the greatest common divisor of two polynomials.

```
gcd(p,q)
```

$$x\ y - x$$

Type: Polynomial Integer

In the case of p and q, the gcd is obvious from their definitions. We factor the gcd to show this relationship better.

```
factor %
```

$$x\ (y - 1)$$

Type: Factored Polynomial Integer

The least common multiple is computed by using **lcm**.

```
lcm(p,q)
```

$$\left(x\ y^2 - 2\ x\ y + x\right)\ z^2 + \left(5\ x\ y^2 - 10\ x\ y + 5\ x\right)\ z$$

Type: Polynomial Integer

Use **content** to compute the greatest common divisor of the coefficients of the polynomial.

```
content p
```

$$1$$

Type: PositiveInteger

Many of the operations on polynomials require you to specify a variable. For example, **resultant** requires you to give the variable in which the polynomials should be expressed.

This computes the resultant of the values of p and q, considering them as polynomials in the variable z. They do not share a root when thought of as polynomials in z.

```
resultant(p,q,z)
```

$$5\ x^2\ y^3 - 15\ x^2\ y^2 + 15\ x^2\ y - 5\ x^2$$

Type: Polynomial Integer

This value is 0 because as polynomials in x the polynomials have a common root.

```
resultant(p,q,x)
```

$$0$$

Type: Polynomial Integer

The data type used for the variables created by `Polynomial` is `Symbol`. As mentioned above, the representation used by `Polynomial` is recursive and so there is a main variable for nonconstant polynomials.

The operation **mainVariable** returns this variable. The return type is actually a union of `Symbol` and `"failed"`.

```
mainVariable p
```

$$z$$

Type: Union(Symbol,...)

The latter branch of the union is be used if the polynomial has no variables, that is, is a constant.

```
mainVariable(1 :: POLY INT)
```

"failed"

Type: Union("failed",...)

You can also use the predicate **ground?** to test whether a polynomial is in fact a member of its ground ring.

```
ground? p
```

false

Type: Boolean

```
ground?(1 :: POLY INT)
```

true

Type: Boolean

The complete list of variables actually used in a particular polynomial is returned by **variables**. For constant polynomials, this list is empty.

```
variables p
```

$$[z, y, x]$$

Type: List Symbol

The **degree** operation returns the degree of a polynomial in a specific variable.

```
degree(p,x)
```

$$1$$

Type: PositiveInteger

```
degree(p,y)
```

$$2$$

Type: PositiveInteger

```
degree(p,z)
```

$$1$$

Type: PositiveInteger

If you give a list of variables for the second argument, a list of the degrees in those variables is returned.

```
degree(p,[x,y,z])
```

$$[1, 2, 1]$$

Type: List NonNegativeInteger

The minimum degree of a variable in a polynomial is computed using **minimumDegree**.

```
minimumDegree(p,z)
```

$$1$$

Type: PositiveInteger

The total degree of a polynomial is returned by **totalDegree**.

```
totalDegree p
```

$$4$$

Type: PositiveInteger

It is often convenient to think of a polynomial as a leading monomial plus the remaining terms.

```
leadingMonomial p
```

$$x\ y^2\ z$$

Type: Polynomial Integer

The **reductum** operation returns a polynomial consisting of the sum of the monomials after the first.

```
reductum p
```

$$(-2\ x\ y + x)\ z$$

Type: Polynomial Integer

These have the obvious relationship that the original polynomial is equal to the leading monomial plus the reductum.

```
p - leadingMonomial p - reductum p
```

$$0$$

Type: Polynomial Integer

The value returned by **leadingMonomial** includes the coefficient of that term. This is extracted by using **leadingCoefficient** on the original polynomial.

```
leadingCoefficient p
```

$$1$$

Type: PositiveInteger

The operation **eval** is used to substitute a value for a variable in a polynomial.

```
p
```

$$\left(x\ y^2 - 2\ x\ y + x\right)\ z$$

Type: Polynomial Integer

This value may be another variable, a constant or a polynomial.

```
eval(p,x,w)
```

$$\left(w\ y^2 - 2\ w\ y + w\right)\ z$$

Type: Polynomial Integer

```
eval(p,x,1)
```

$$\left(y^2 - 2\ y + 1\right)\ z$$

Type: Polynomial Integer

Actually, all the things being substituted are just polynomials, some more trivial than others.

```
eval(p,x,y**2 - 1)
```

$$\left(y^4 - 2\ y^3 + 2\ y - 1\right)\ z$$

Type: Polynomial Integer

Derivatives are computed using the **D** operation.

```
D(p,x)
```

$$\left(y^2 - 2\ y + 1\right)\ z$$

Type: Polynomial Integer

The first argument is the polynomial and the second is the variable.

```
D(p,y)
```

$$(2\ x\ y - 2\ x)\ z$$

Type: Polynomial Integer

Even if the polynomial has only one variable, you must specify it.

```
D(p,z)
```

$$x\ y^2 - 2\ x\ y + x$$

Type: Polynomial Integer

Integration of polynomials is similar and the **integrate** operation is used.

Integration requires that the coefficients support division. Consequently, AXIOM converts polynomials over the integers to polynomials over the rational numbers before integrating them.

```
integrate(p,y)
```

$$\left(\frac{1}{3}\ x\ y^3 - x\ y^2 + x\ y\right)\ z$$

Type: Polynomial Fraction Integer

It is not possible, in general, to divide two polynomials. In our example using polynomials over the integers, the operation **monicDivide** divides a polynomial by a monic polynomial (that is, a polynomial with leading coefficient equal to 1). The result is a record of the quotient and remainder of the division.

You must specify the variable in which to express the polynomial.

```
qr := monicDivide(p,x+1,x)
```

$$\left[quotient = \left(y^2 - 2\ y + 1\right)\ z, remainder = \left(-y^2 + 2\ y - 1\right)\ z\right]$$

Type: Record(quotient: Polynomial Integer,remainder: Polynomial Integer)

The selectors of the components of the record are `quotient` and `remainder`. Issue this to extract the remainder.

```
qr.remainder
```

$$\left(-y^2 + 2\ y - 1\right)\ z$$

Type: Polynomial Integer

Now that we can extract the components, we can demonstrate the relationship among them and the arguments to our original expression `qr := monicDivide(p,x+1,x)`.

```
p - ((x+1) * qr.quotient + qr.remainder)
```

$$0$$

Type: Polynomial Integer

If the "/" operator is used with polynomials, a fraction object is created. In this example, the result is an object of type `Fraction Polynomial Integer`.

```
p/q
```

$$\frac{(y-1)\ z}{z+5}$$

Type: Fraction Polynomial Integer

If you use rational numbers as polynomial coefficients, the resulting object is of type `Polynomial Fraction Integer`.

```
(2/3) * x**2 - y + 4/5
```

$$-y + \frac{2}{3}\ x^2 + \frac{4}{5}$$

Type: Polynomial Fraction Integer

This can be converted to a fraction of polynomials and back again, if required.

```
% :: FRAC POLY INT
```

$$\frac{-15\ y + 10\ x^2 + 12}{15}$$

Type: Fraction Polynomial Integer

```
% :: POLY FRAC INT
```

$$-y + \frac{2}{3}\ x^2 + \frac{4}{5}$$

Type: Polynomial Fraction Integer

To convert the coefficients to floating point, map the `numeric` operation on the coefficients of the polynomial.

```
map(numeric,%)
```

$$-1.0\ y + 0.66666666666666666667\ x^2 + 0.8$$

Type: Polynomial Float

For more information on related topics, see 9.67 on page 225, 9.41 on page 153, and 9.15 on page 59. You can also issue the system command `)show Polynomi- al` to display the full list of operations defined by `Polynomial`.

9.50 Quaternion

The domain constructor `Quaternion` implements quaternions over commutative rings. For information on related topics, see 9.10 on page 34 and 9.43 on page 156. You can also issue the system command `)show Quaternion` to display the full list of operations defined by `Quaternion`.

The basic operation for creating quaternions is **quatern**. This is a quaternion over the rational numbers.

```
q := quatern(2/11,-8,3/4,1)
```

$$\frac{2}{11} - 8\ i + \frac{3}{4}\ j + k$$

Type: Quaternion Fraction Integer

The four arguments are the real part, the `i` imaginary part, the `j` imaginary part, and the `k` imaginary part, respectively.

```
[real q, imagI q, imagJ q, imagK q]
```

$$\left[\frac{2}{11}, -8, \frac{3}{4}, 1\right]$$

Type: List Fraction Integer

Because `q` is over the rationals (and nonzero), you can invert it.

```
inv q
```

$$\frac{352}{126993} + \frac{15488}{126993}\ i - \frac{484}{42331}\ j - \frac{1936}{126993}\ k$$

Type: Quaternion Fraction Integer

The usual arithmetic (ring) operations are available

```
q**6
```

$$-\frac{2029490709319345}{7256313856} - \frac{48251690851}{1288408}\ i + \frac{144755072553}{41229056}\ j + \frac{48251690851}{10307264}\ k$$

Type: Quaternion Fraction Integer

```
r := quatern(-2,3,23/9,-89); q + r
```

$$-\frac{20}{11} - 5\ i + \frac{119}{36}\ j - 88\ k$$

Type: Quaternion Fraction Integer

In general, multiplication is not commutative.

```
q * r - r * q
```

$$-\frac{2495}{18}\ i - 1418\ j - \frac{817}{18}\ k$$

Type: Quaternion Fraction Integer

There are no predefined constants for the imaginary `i, j`, and `k` parts, but you can easily define them.

```
i:=quatern(0,1,0,0); j:=quatern(0,0,1,0); k:=quatern(0,0,0,1)
```

$$k$$

Type: Quaternion Integer

These satisfy the normal identities.

```
[i*i, j*j, k*k, i*j, j*k, k*i, q*i]
```

$$\left[-1, -1, -1, k, i, j, 8 + \frac{2}{11}\ i + j - \frac{3}{4}\ k\right]$$

Type: List Quaternion Fraction Integer

The norm is the quaternion times its conjugate.

```
norm q
```

$$\frac{126993}{1936}$$

Type: Fraction Integer

```
conjugate q
```

$$\frac{2}{11} + 8\ i - \frac{3}{4}\ j - k$$

Type: Quaternion Fraction Integer

```
q * %
```

$$\frac{126993}{1936}$$

Type: Quaternion Fraction Integer

9.51 RadixExpansion

It possible to expand numbers in general bases.

Here we expand `111` in base `5`. This means

$$10^2 + 10^1 + 10^0 = 4 \cdot 5^2 + 2 \cdot 5^1 + 5^0$$

```
111::RadixExpansion(5)
```

Type: RadixExpansion 5

You can expand fractions to form repeating expansions.

```
(5/24)::RadixExpansion(2)
```

$$0.001\overline{10}$$

Type: RadixExpansion 2

```
(5/24)::RadixExpansion(3)
```

$$0.0\overline{12}$$

Type: RadixExpansion 3

```
(5/24)::RadixExpansion(8)
```

$$0.1\overline{52}$$

Type: RadixExpansion 8

```
(5/24)::RadixExpansion(10)
```

$$0.208\overline{3}$$

Type: RadixExpansion 10

For bases from 11 to 36 the letters A through Z are used.

```
(5/24)::RadixExpansion(12)
```

$$0.26$$

Type: RadixExpansion 12

```
(5/24)::RadixExpansion(16)
```

$$0.3\overline{5}$$

Type: RadixExpansion 16

```
(5/24)::RadixExpansion(36)
```

$$0.7I$$

Type: RadixExpansion 36

For bases greater than 36, the ragits are separated by blanks.

```
(5/24)::RadixExpansion(38)
```

$$0\ .\ 7\ 34\ 31\ \overline{25\ 12}$$

Type: RadixExpansion 38

The `RadixExpansion` type provides operations to obtain the individual ragits. Here is a rational number in base 8.

```
a := (76543/210)::RadixExpansion(8)
```

$$554.3\overline{7307}$$

Type: RadixExpansion 8

The operation **wholeRagits** returns a list of the ragits for the integral part of the number.

```
w := wholeRagits a
```

$$[5, 5, 4]$$

Type: List Integer

The operations **prefixRagits** and **cycleRagits** return lists of the initial and repeating ragits in the fractional part of the number.

```
f0 := prefixRagits a
```

$$[3]$$

Type: List Integer

```
f1 := cycleRagits a
```

$$[7, 3, 0, 7]$$

Type: List Integer

You can construct any radix expansion by giving the whole, prefix and cycle parts. The declaration is necessary to let AXIOM know the base of the ragits.

```
u:RadixExpansion(8):=wholeRadix(w)+fractRadix(f0,f1)
```

$$554.3\overline{7307}$$

Type: RadixExpansion 8

If there is no repeating part, then the list [0] should be used.

```
v:  RadixExpansion(12) := fractRadix([1,2,3,11], [0])
```

$$0.123\mathrm{B}\overline{0}$$

Type: RadixExpansion 12

If you are not interested in the repeating nature of the expansion, an infinite stream of ragits can be obtained using **fractRagits**.

```
fractRagits(u)
```

$$\left[3, 7, \overline{3, 0, 7, 7}\right]$$

Type: Stream Integer

Of course, it's possible to recover the fraction representation:

```
a ::  Fraction(Integer)
```

$$\frac{76543}{210}$$

Type: Fraction Integer

Issue the system command `) show RadixExpansion` to display the full list of operations defined by `RadixExpansion`. More examples of expansions are available in 9.14 on page 58, 9.3 on page 6, and 9.29 on page 96.

9.52 RomanNumeral

The Roman numeral package was added to AXIOM in MCMLXXXVI for use in denoting higher order derivatives.

For example, let `f` be a symbolic operator.

```
f := operator 'f
```

$$f$$

Type: BasicOperator

This is the seventh derivative of `f` with respect to `x`.

```
D(f x,x,7)
```

$$f^{(vii)}(x)$$

Type: Expression Integer

You can have integers printed as Roman numerals by declaring variables to be of type `RomanNumeral` (abbreviation `ROMAN`).

```
a := roman(1978 - 1965)
```

$$XIII$$

Type: RomanNumeral

This package now has a small but devoted group of followers that claim this domain has shown its efficacy in many other contexts. They claim that Roman numerals are every bit as useful as ordinary integers.

In a sense, they are correct, because Roman numerals form a ring and you can therefore construct polynomials with Roman numeral coefficients, matrices over Roman numerals, etc..

```
x : UTS(ROMAN,'x,0) := x
```

$$x$$

Type: UnivariateTaylorSeries(RomanNumeral,x,0)

Was Fibonacci Italian or ROMAN?

```
recip(1 - x - x**2)
```

$$I + x + II\ x^2 + III\ x^3 + V\ x^4 + VIII\ x^5 + XIII\ x^6 + XXI\ x^7 + O\left(x^8\right)$$

Type: Union(UnivariateTaylorSeries(RomanNumeral,x,0),...)

You can also construct fractions with Roman numeral numerators and denominators, as this matrix Hilberticus illustrates.

```
m : MATRIX FRAC ROMAN
```

Void

```
m := matrix [ [1/(i + j) for i in 1..3] for j in 1..3]
```

$$\begin{bmatrix} \frac{I}{II} & \frac{I}{III} & \frac{I}{IV} \\ \frac{I}{III} & \frac{I}{IV} & \frac{I}{V} \\ \frac{I}{IV} & \frac{I}{V} & \frac{I}{VI} \end{bmatrix}$$

Type: Matrix Fraction RomanNumeral

Note that the inverse of the matrix has integral `ROMAN` entries.

```
inverse m
```

$$\begin{bmatrix} LXXII & -CCXL & CLXXX \\ -CCXL & CM & -DCCXX \\ CLXXX & -DCCXX & DC \end{bmatrix}$$

Type: Union(Matrix Fraction RomanNumeral,...)

Unfortunately, the spoil-sports say that the fun stops when the numbers get big—mostly because the Romans didn't establish conventions about representing very large numbers.

```
y := factorial 10
```

3628800

Type: PositiveInteger

You work it out!

```
roman y
```

((((I))))((((I))))((((I))))(((I)))(((I)))(((I)))(((I)))
(((I)))(((I)))((I))((I))MMMMMMMMDCCC

Type: RomanNumeral

Issue the system command `)show RomanNumeral` to display the full list of operations defined by `RomanNumeral`.

9.53 Segment

The `Segment` domain provides a generalized interval type.

Segments are created using the "`..`" construct by indicating the (included) end points.

```
s := 3..10
```

3..10

Type: Segment PositiveInteger

The first end point is called the **lo** and the second is called **hi**.

```
lo s
```

3

Type: PositiveInteger

These names are used even though the end points might belong to an unordered set.

```
hi s
```

$$10$$

Type: PositiveInteger

In addition to the end points, each segment has an integer "increment." An increment can be specified using the "`by`" construct.

```
t := 10..3 by -2
```

$$10..3 \text{ by } -2$$

Type: Segment PositiveInteger

This part can be obtained using the **incr** function.

```
incr s
```

$$1$$

Type: PositiveInteger

Unless otherwise specified, the increment is `1`.

```
incr t
```

$$-2$$

Type: Integer

A single value can be converted to a segment with equal end points. This happens if segments and single values are mixed in a list.

```
l := [1..3, 5, 9, 15..11 by -1]
```

$$[1..3, 5..5, 9..9, 15..11 \text{by } -1]$$

Type: List Segment PositiveInteger

If the underlying type is an ordered ring, it is possible to perform additional operations. The **expand** operation creates a list of points in a segment.

```
expand s
```

$$[3, 4, 5, 6, 7, 8, 9, 10]$$

Type: `List Integer`

If `k > 0`, then `expand(l..h by k)` creates the list `[l, l+k, ..., lN]` where `lN <= h < lN+k`. If `k < 0`, then `lN >= h > lN+k`.

```
expand t
```

$$[10, 8, 6, 4]$$

Type: `List Integer`

It is also possible to expand a list of segments. This is equivalent to appending lists obtained by expanding each segment individually.

```
expand l
```

$$[1, 2, 3, 5, 9, 15, 14, 13, 12, 11]$$

Type: `List Integer`

For more information on related topics, see 9.54 on page 191 and 9.68 on page 232. Issue the system command `)show Segment` to display full list of operations defined by `Segment`.

9.54 SegmentBinding

The `SegmentBinding` type is used to indicate a range for a named symbol.

First give the symbol, then an "`=`" and finally a segment of values.

```
x = a..b
```

$$x = a..b$$

Type: `SegmentBinding Symbol`

This is used to provide a convenient syntax for arguments to certain operations.

```
sum(i**2, i = 0..n)
```

$$\frac{2\ n^3 + 3\ n^2 + n}{6}$$

Type: `Fraction Polynomial Integer`

The **draw** operation uses a `SegmentBuilding` argument as a range of coordinates. This is an example of a two-dimensional parameterized plot; other **draw** options use more than one `SegmentBuilding` argument.

```
draw(x**2, x = -2..2)
```

The left-hand side must be of type `Symbol` but the right-hand side can be a segment over any type.

```
sb := y = 1/2..3/2
```

$$y = \left(\frac{1}{2}\right)..\left(\frac{3}{2}\right)$$

Type: SegmentBinding Fraction Integer

The left- and right-hand sides can be obtained using the **variable** and **segment** operations.

```
variable(sb)
```

$$y$$

Type: Symbol

```
segment(sb)
```

$$\left(\frac{1}{2}\right)..\left(\frac{3}{2}\right)$$

Type: Segment Fraction Integer

Issue the system command `)show SegmentBinding` to display the full list of operations defined by `SegmentBinding`. For more information on related topics, see 9.53 on page 189 and 9.68 on page 232.

9.55 Set

The `Set` domain allows one to represent explicit finite sets of values. These are similar to lists, but duplicate elements are not allowed.

Sets can be created by giving a fixed set of values ...

```
s := set [x**2-1, y**2-1, z**2-1]
```

$$\left\{x^2-1, y^2-1, z^2-1\right\}$$

Type: Set Polynomial Integer

or by using a collect form, just as for lists. In either case, the set is formed from a finite collection of values.

```
t := set [x**i - i+1 for i in 2..10 | prime?  i]
```

$$\left\{x^2-1, x^3-2, x^5-4, x^7-6\right\}$$

Type: Set Polynomial Integer

The basic operations on sets are **intersect**, **union**, **difference**, and **symmetricDifference**.

```
i := intersect(s,t)
```

$$\left\{x^2-1\right\}$$

Type: Set Polynomial Integer

```
u := union(s,t)
```

$$\left\{x^2-1, x^3-2, x^5-4, x^7-6, y^2-1, z^2-1\right\}$$

Type: Set Polynomial Integer

The set `difference(s,t)` contains those members of `s` which are not in `t`.

```
difference(s,t)
```

$$\left\{y^2-1, z^2-1\right\}$$

Type: Set Polynomial Integer

The set `symmetricDifference(s,t)` contains those elements which are in `s` or `t` but not in both.

```
symmetricDifference(s,t)
```

$$\left\{x^3-2, x^5-4, x^7-6, y^2-1, z^2-1\right\}$$

Type: Set Polynomial Integer

Set membership is tested using the **member?** operation.

```
member?(y, s)
```

false

Type: Boolean

```
member?((y+1)*(y-1), s)
```

true

Type: Boolean

The **subset?** function determines whether one set is a subset of another.

```
subset?(i, s)
```

true

Type: Boolean

```
subset?(u, s)
```

false

Type: Boolean

When the base type is finite, the absolute complement of a set is defined. This finds the set of all multiplicative generators of `PrimeField 11`—the integers mod `11`.

```
gs := set [g for i in 1..11 | primitive?(g := i::PF 11)]
```

$$\{2, 6, 7, 8\}$$

Type: Set PrimeField 11

The following values are not generators.

```
complement gs
```

$$\{1, 3, 4, 5, 9, 10, 0\}$$

Type: Set PrimeField 11

Often the members of a set are computed individually; in addition, values can be inserted or removed from a set over the course of a computation.

There are two ways to do this:

```
a := set [i**2 for i in 1..5]
```

$$\{1, 4, 9, 16, 25\}$$

Type: Set PositiveInteger

One is to view a set as a data structure and to apply updating operations.

```
insert!(32, a)
```

$$\{1, 4, 9, 16, 25, 32\}$$

Type: Set PositiveInteger

```
remove!(25, a)
```

$$\{1, 4, 9, 16, 32\}$$

Type: Set PositiveInteger

```
a
```

$$\{1, 4, 9, 16, 32\}$$

Type: Set PositiveInteger

The other way is to view a set as a mathematical entity and to create new sets from old.

```
b := b0 := set [i**2 for i in 1..5]
```

$$\{1, 4, 9, 16, 25\}$$

Type: Set PositiveInteger

```
b := union(b, {32})
```

$$\{1, 4, 9, 16, 25, 32\}$$

Type: Set PositiveInteger

```
b := difference(b, {25})
```

$$\{1, 4, 9, 16, 32\}$$

Type: Set PositiveInteger

```
b0
```

$$\{1, 4, 9, 16, 25\}$$

Type: Set PositiveInteger

For more information about lists, see 9.36 on page 129. Issue the system command `)show Set` to display the full list of operations defined by `Set`.

9.56 SmallFloat

AXIOM provides two kinds of floating point numbers. The domain `Float` (abbreviation `FLOAT`) implements a model of arbitrary precision floating point numbers. The domain `SmallFloat` (abbreviation `SF`) is intended to make available hardware floating point arithmetic in AXIOM. The actual model of floating point `SmallFloat` that provides is system-dependent. For example, on the IBM system 370 AXIOM uses IBM double precision which has fourteen hexadecimal digits of precision or roughly sixteen decimal digits. Arbitrary precision floats allow the user to specify the precision at which arithmetic operations are computed. Although this is an attractive facility, it comes at a cost. Arbitrary-precision floating-point arithmetic typically takes twenty to two hundred times more time than hardware floating point.

Th usual arithmetic and elementary functions are available for `SmallFloat`. Use `)show SmallFloat` to get a list of operations or the `HyperDoc Browse` facility to get more extensive documentation about `SmallFloat`.

By default, floating point numbers that you enter into AXIOM are of type `Float`.

```
2.71828
```

$$2.71828$$

`Type: Float`

You must therefore tell AXIOM that you want to use `SmallFloat` values and operations. The following are some conservative guidelines for getting AXIOM to use `SmallFloat`.

To get a value of type `SmallFloat`, use a target with "`@`",...

```
2.71828@SmallFloat
```

$$2.71828$$

`Type: SmallFloat`

a conversion, ...

```
2.71828 :: SmallFloat
```

`Type: SmallFloat`

or an assignment to a declared variable. It is more efficient if you use a target rather than an explicit or implicit conversion.

```
eApprox : SmallFloat := 2.71828
```

$$2.71828$$

Type: SmallFloat

You also need to declare functions that work with `SmallFloat`.

```
avg : List SmallFloat -> SmallFloat
```

Type: Void

```
avg l ==

  empty? l => 0 :: SmallFloat

reduce(_+,l) / #l
```

Type: Void

```
avg []

Compiling function avg with type List SmallFloat ->
  SmallFloat
```

$$0.0$$

Type: SmallFloat

```
avg [3.4,9.7,-6.8]
```

$$2.1000000000000001$$

Use package calling for operations from `SmallFloat` unless the arguments themselves are already of type `SmallFloat`.

```
cos(3.1415926)$SmallFloat
```

$$-0.99999999999999856$$

Type: SmallFloat

```
cos(3.1415926 :: SmallFloat)
```

$$-0.99999999999999856$$

Type: SmallFloat

By far, the most common usage of `SmallFloat` is for functions to be graphed. For more information about AXIOM's numerical and graphical facilities, see Section 9.24 on page 82.

9.57 SmallInteger

The `SmallInteger` domain is intended to provide support in AXIOM for machine integer arithmetic. It is generally much faster than (bignum) `Integer` arithmetic but suffers from a limited range of values. Since AXIOM can be implemented on top of varios aspects of Lisp, the actual representation of small integers may not correspond exactly to the host machines integer representation.

Under *AKCL* on the IBM RISC System/6000, small integers are restricted to the range -2^{26} to $2^{26}-1$, allowing 1bit for overflow detection.

You can discover the minimum and maximum values in your implementation by using **min** and **max**.

```
min()$SmallInteger
```

$$-2147483648$$

Type: `SmallInteger`

```
max()$SmallInteger
```

$$2147483647$$

Type: `SmallInteger`

To avoid confusion with `Integer`, which is the default type for integers, you usually need to work with declared variables ...

```
a := 1234 ::  SmallInteger
```

$$1234$$

Type: `SmallInteger`

or use package calling.

```
b := 124$SmallInteger
```

$$124$$

Type: `SmallInteger`

You can add, multiply and subtract `SmallInteger` objects, and ask for the greatest common divisor (**gcd**).

```
gcd(a,b)
```

$$2$$

Type: `SmallInteger`

The least common multiple (**lcm**) is also available.

```
lcm(a,b)
```

$$76508$$

Type: `SmallInteger`

Operations **nullmod**, **addmod**, **submod** and **invmod** are similar - they provide arithmetic modulo a given small integer. Here is `5 * 6 mod 13`.

```
mulmod(5,6,13)$SmallInteger
```

$$4$$

Type: `SmallInteger`

To reduce a small integer modulo a prime, use **positiveRemainder**.

```
positiveRemainder(37,13)$SmallInteger
```

$$11$$

Type: `SmallInteger`

Operations **And**, **Or**, **xor**, and **Not** provide bit level operations on small integers.

```
And (3,4)$SmallInteger
```

$$0$$

Type: `SmallInteger`

Use `shift (int,numToShift)` to shift bits, where `i` is shifted left if `numToShift` is positive, right if negative.

```
shift(1,4)$SmallInteger
```

$$16$$

Type: `SmallInteger`

```
shift(31,-1)$SmallInteger
```

$$15$$

Type: `SmallInteger`

Many other operations are available for small integers, including many of those provided for `Integer`. To see the other operations, use the `Browse HyperDoc` facility. Issue the system command `)show SmallInteger` to display the full list of operations defined by `SmallInteger`.

9.58 SparseTable

The `SparseTable` domain provides a general purpose table type with default entries.

Here we create a table to save strings under integer keys. The value `"Try again!"` is returned if no other value has been stored for a key.

```
t:  SparseTable(Integer, String, "Try again!") := table()
```

$$table()$$

Type: `SparseTable(Integer,String,Try again!)`

Entries can be stored in the table.

```
t.3 := "Number three"
```

"Number three"

Type: `String`

```
t.4 := "Number four"
```

"Number four"

Type: `String`

These values can be retrieved as usual, but if a look up fails the default entry will be returned.

```
t.3
```

"Number three"

Type: `String`

```
t.2
```

"Try again!"

Type: `String`

To see which values are explicitly stored, the **keys** and **entries** functions can be used.

```
keys t
```

$$[4, 3]$$

Type: List Integer

```
entries t
```

["Number four","Number three"]

Type: List String

If a specific table representation is required, the `GeneralSparseTable` constructor should be used. The domain `SparseTable(K, E, dflt)` is equivalent to `GeneralSparseTable(K,E,Table(K,E), dflt)`. For more information, see 9.64 on page 215 and 9.26 on page 91. Issue the system command `)show SparseTable` to display the full list of operations defined by `SparseTable`.

9.59 SquareMatrix

The top level matrix type in AXIOM is `Matrix` (see 9.39 on page 142), which provides basic arithmetic and linear algebra functions. However, since the matrices can be of any size it is not true that any pair can be added or multiplied. Thus `Matrix` has little algebraic structure.

Sometimes you want to use matrices as coefficients for polynomials or in other algebraic contexts. In this case, `SquareMatrix` should be used. The domain `SquareMatrix(n,R)` gives the ring of `n` by `n` square matrices over `R`.

Since `SquareMatrix` is not normally exposed at the top level, you must expose it before it can be used.

```
)set expose add constructor SquareMatrix
```

```
   SquareMatrix is now explicitly exposed in frame G1077
```

Once `SQMATRIX` has been exposed, values can be created using the **squareMatrix** function.

```
m := squareMatrix [ [1,-%i],[%i,4] ]
```

$$\begin{bmatrix} 1 & -\%i \\ \%i & 4 \end{bmatrix}$$

Type: SquareMatrix(2,Complex Integer)

The usual arithmetic operations are available.

```
m*m - m
```

$$\begin{bmatrix} 1 & -4\ \%i \\ 4\ \%i & 13 \end{bmatrix}$$

Type: SquareMatrix(2,Complex Integer)

Square matrices can be used where ring elements are required. For example, here is a matrix with matrix entries.

```
mm := squareMatrix [ [m, 1], [1-m, m**2] ]
```

$$\begin{bmatrix} \begin{bmatrix} 1 & -\%i \\ \%i & 4 \end{bmatrix} & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ \begin{bmatrix} 0 & \%i \\ -\%i & -3 \end{bmatrix} & \begin{bmatrix} 2 & -5\ \%i \\ 5\ \%i & 17 \end{bmatrix} \end{bmatrix}$$

Type: SquareMatrix(2,SquareMatrix(2,Complex Integer))

Or you can construct a polynomial with square matrix coefficients.

```
p := (x + m)**2
```

$$x^2 + \begin{bmatrix} 2 & -2\ \%i \\ 2\ \%i & 8 \end{bmatrix} x + \begin{bmatrix} 2 & -5\ \%i \\ 5\ \%i & 17 \end{bmatrix}$$

Type: Polynomial SquareMatrix(2,Complex Integer)

This value can be converted to a square matrix with polynomial coefficients.

```
p::SquareMatrix(2, ?)
```

$$\begin{bmatrix} x^2 + 2\ x + 2 & -2\ \%i\ x - 5\ \%i \\ 2\ \%i\ x + 5\ \%i & x^2 + 8\ x + 17 \end{bmatrix}$$

Type: SquareMatrix(2,Polynomial Complex Integer)

For more information on related topics, see Section 9.39 on page 142. Issue the system command `)show SquareMatrix` to display the full list of operations defined by `SquareMatrix`.

9.60 Stream

A `Stream` object is represented as a list whose last element contains the wherewithal to create the next element, should it ever be required.

Let `ints` be the infinite stream of non-negative integers.

```
ints := [i for i in 0..]
```

$$[0, 1, 2, 3, 4, 5, 6, \ldots]$$

Type: Stream NonNegativeInteger

By default, ten stream elements are calculated. This number may be changed to something else by the system command `)set streams calculate`. For the display purposes of this book, we have chosen a smaller value.

More generally, you can construct a stream by specifying its initial value and a function which, when given an element, creates the next element.

```
f : List INT -> List INT
```

Void

```
f x == [x.1 + x.2, x.1]
```

Void

```
fibs := [i.2 for i in [generate(f,[1,1])]]
```

```
   Compiling function f with type List Integer -> List Integer
```

$$[1, 1, 2, 3, 5, 8, 13, \ldots]$$

Type: Stream Integer

You can create the stream of odd non-negative integers by either filtering them from the integers, or by evaluating an expression for each integer.

```
[i for i in ints | odd? i]
```

$$[1, 3, 5, 7, 9, 11, 13, \ldots]$$

Type: Stream NonNegativeInteger

```
odds := [2*i+1 for i in ints]
```

$$[1, 3, 5, 7, 9, 11, 13, \ldots]$$

Type: Stream NonNegativeInteger

You can accumulate the initial segments of a stream using the **scan** operation.

```
scan(0,+,odds)
```

$$[1, 4, 9, 16, 25, 36, 49, \ldots]$$

Type: Stream NonNegativeInteger

The corresponding elements of two or more streams can be combined in this way.

```
[i*j for i in ints for j in odds]
```

$$[0, 3, 10, 21, 36, 55, 78, \ldots]$$

Type: Stream NonNegativeInteger

```
map(*,ints,odds)
```

$$[0, 3, 10, 21, 36, 55, 78, \ldots]$$

Type: Stream NonNegativeInteger

Many operations similar to those applicable to lists are available for streams.

```
first ints
```

$$0$$

Type: NonNegativeInteger

```
rest ints
```

$$[1, 2, 3, 4, 5, 6, 7, \ldots]$$

Type: Stream NonNegativeInteger

```
fibs 20
```

$$6765$$

Type: PositiveInteger

The packages `StreamFunctions1`, `StreamFunctions2` and `StreamFuncti-ons3` export some useful stream manipulation operations. For more information, see Section 9.11 on page 37, and 9.36 on page 129. Issue the system command `)show Stream` to display the full list of operations defined by `Stream`.

9.61 String

The type `String` provides character strings. Character strings provide all the operations for a one-dimensional array of characters, plus additional operations for manipulating text. For more information on related topics, see 9.7 on page 23 and 9.8 on page 25. You can also issue the system command `)show String` to display the full list of operations defined by `String`.

String values can be created using double quotes.

```
hello := "Hello, I'm AXIOM!"
```

```
"Hello, I'm AXIOM!"
```

Type: String

Note, however, that double quotes and underscores must be preceded by an extra underscore.

```
said := "Jane said, _"Look!_""
```

```
"Jane said, "Look!""
```

Type: String

```
saw := "She saw exactly one underscore:  __."
```

```
"She saw exactly one underscore:  _."
```

Type: String

It is also possible to use **new** to create a string of any size filled with a given character. Since there are many `new` functions it is necessary to indicate the desired type.

```
gasp:  String := new(32, char "x")
```

```
"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
```

Type: String

The length of a string is given by "`#`".

```
#gasp
```

```
32
```

Type: PositiveInteger

Indexing operations allow characters to be extracted or replaced in strings. For any string `s`, indices lie in the range `1..#s`.

```
hello.2
```

$$e$$

Type: Character

Indexing is really just the application of a string to a subscript, so any application syntax works.

```
hello 2
```

$$e$$

Type: Character

```
hello(2)
```

$$e$$

Type: Character

If it is important not to modify a given string, it should be copied before any updating operations are used.

```
hullo := copy hello
```

"Hello, I'm AXIOM!"

Type: String

```
hullo.2 := char "u"; [hello, hullo]
```

["Hello, I'm AXIOM!","Hullo, I'm AXIOM!"]

Type: List String

Operations are provided to split and join strings. The **concat** operation allows several strings to be joined together.

```
saidsaw := concat ["alpha","--","omega"]
```

"alpha--omega"

Type: String

There is a version of **concat** that works with two strings.

```
concat("hello ","goodbye")
```

"hello goodbye"

Type: String

Juxtaposition can also be used to concatenate strings.

```
"This " "is " "several " "strings " "concatenated."
```

"This is several strings concatenated."

Type: String

Substrings are obtained by giving an index range.

```
hello(1..5)
```

"Hello"

Type: String

```
hello(8..)
```

"I'm AXIOM!"

Type: String

A string can be split into several substrings by giving a separation character or character class.

```
split(hello, char " ")
```

["Hello,","I'm","AXIOM!"]

Type: List String

```
other := complement alphanumeric();
```

Type: CharacterClass

```
split(saidsaw, other)
```

["alpha","omega"]

Type: List String

Unwanted characters can be trimmed from the beginning or end of a string using the operations **trim**, **leftTrim** and **rightTrim**.

```
trim("## ++ relax ++ ##", char "#")
```

```
" ++ relax ++ "
```

```
Type: String
```

Each of these functions takes a string and a second argument to specify the characters to be discarded.

```
trim("## ++ relax ++ ##", other)
```

```
"relax"
```

```
Type: String
```

The second argument can be given either as a single character or as a character class.

```
leftTrim ("## ++ relax ++ ##", other)
```

```
"relax ++ ##"
```

```
Type: String
```

```
rightTrim("## ++ relax ++ ##", other)
```

```
"## ++ relax"
```

```
Type: String
```

Strings can be changed to upper case or lower case using the operations **upperCase**, **upperCase**, **lowerCase** and **lowerCase**.

```
upperCase hello
```

```
"HELLO, I'M AXIOM!"
```

```
Type: String
```

The versions with the exclamation mark change the original string, while the others produce a copy.

```
lowerCase hello
```

```
"hello, i'm axiom!"
```

Type: String

Some basic string matching is provided. The function **prefix?** tests whether one string is an initial prefix of another.

```
prefix?("He", "Hello")
```

true

Type: Boolean

```
prefix?("Her", "Hello")
```

false

Type: Boolean

A similar function, **suffix?**, tests for suffixes.

```
suffix?("", "Hello")
```

true

Type: Boolean

```
suffix?("LO", "Hello")
```

false

Type: Boolean

The function **substring?** tests for a substring given a starting position.

```
substring?("ll", "Hello", 3)
```

true

Type: Boolean

```
substring?("ll", "Hello", 4)
```

false

Type: Boolean

A number of **position** functions locate things in strings. If the first argument to position is a string, then `position(s,t,i)` finds the location of `s` as a substring of `t` starting the search at position `i`.

```
n := position("nd", "underground", 1)
```

$$2$$

Type: PositiveInteger

```
n := position("nd", "underground", n+1)
```

$$10$$

Type: PositiveInteger

If `s` is not found, then `0` is returned (`minIndex(s)-1` in `IndexedString`).

```
n := position("nd", "underground", n+1)
```

$$0$$

Type: NonNegativeInteger

To search for a specific character or a member of a character class, a different first argument is used.

```
position(char "d", "underground", 1)
```

$$3$$

Type: PositiveInteger

```
position(hexDigit(), "underground", 1)
```

$$3$$

Type: PositiveInteger

9.62 StringTable

This domain provides a table type in which the keys are known to be strings so special techniques can be used. Other than performance, the type `StringTable (S)` should behave exactly the same way as `Table(String,S)`. See 9.64 on page 215 for general information about tables. Issue the system command `)show StringTable` to display the full list of operations defined by `StringTable`.

This creates a new table whose keys are strings.

```
t:  StringTable(Integer) := table()
```

$$table()$$

Type: StringTable Integer

The value associated with each string key is the number of characters in the string.

```
for s in split("My name is Ian Watt.",char " ")
  repeat
    t.s := #s
```

Type: Void

```
for key in keys t repeat output [key, t.key]
```

```
   ["Ian",3]
   ["My",2]
   ["Watt.",5]
   ["name",4]
   ["is",2]
```

Type: Void

9.63 Symbol

Symbols are one of the basic types manipulated by AXIOM. The `Symbol` domain provides ways to create symbols of many varieties. Issue the system command `)show Symbol` to display the full list of operations defined by `Symbol`.

The simplest way to create a symbol is to "single quote" an identifier.

```
X: Symbol := 'x
```

$$x$$

Type: Symbol

This gives the symbol even if `x` has been assigned a value. If `x` has not been assigned a value, then it is possible to omit the quote.

```
XX: Symbol := x
```

$$x$$

Type: Symbol

Declarations must be used when working with symbols, because otherwise the interpreter tries to place values in a more specialized type `Variable`.

```
A := 'a
```

$$a$$

Type: Variable a

```
B := b
```

$$b$$

Type: Variable b

The normal way of entering polynomials uses this fact.

```
x**2 + 1
```

$$x^2 + 1$$

Type: Polynomial Integer

Another convenient way to create symbols is to convert a string. This is useful when the name is to be constructed by a program.

```
"Hello"::Symbol
```

$$Hello$$

Type: Symbol

Sometimes it is necessary to generate new unique symbols, for example, to name constants of integration. The expression `new()` generates a symbol starting with `%`.

```
new()$Symbol
```

$$\%A$$

Type: Symbol

Successive calls to **new** produce different symbols.

```
new()$Symbol
```

$$\%B$$

Type: Symbol

The expression `new("s")` produces a symbol starting with `%s`.

```
new("xyz")$Symbol
```

$$\%xyz0$$

Type: Symbol

A symbol can be adorned in various ways. The most basic thing is applying a symbol to a list of subscripts.

```
X[i,j]
```

$$x_{i,j}$$

Type: Symbol

Somewhat less pretty is to attach subscripts, superscripts or arguments.

```
U := subscript(u, [1,2,1,2])
```

$$u_{1,2,1,2}$$

Type: Symbol

```
V := superscript(v, [n])
```

$$v^{n}$$

Type: Symbol

```
P := argscript(p, [t])
```

$$p\,(t)$$

Type: Symbol

It is possible to test whether a symbol has scripts using the **scripted?** test.

```
scripted? U
```

true

Type: Boolean

```
scripted? X
```

false

Type: Boolean

If a symbol is not scripted, then it may be converted to a string.

```
string X
```

"x"

Type: String

The basic parts can always be extracted using the **name** and **scripts** operations.

```
name U
```

$$u$$

Type: Symbol

```
scripts U
```

$$[sub = [1,2,1,2], sup = [], presup = [], presub = [], args = []]$$

Type: Record(sub: List OutputForm, sup: List OutputForm, presup: List OutputForm, presub: List OutputForm, args: List OutputForm)

```
name X
```

$$x$$

Type: Symbol

```
scripts X
```

$$[sub = [], sup = [], presup = [], presub = [], args = []]$$

Type: Record(sub: List OutputForm, sup: List OutputForm, presup: List OutputForm, presub: List OutputForm, args: List OutputForm)

The most general form is obtained using the **script** operation. This operation takes an argument which is a list containing, in this order, lists of subscripts, superscripts, presuperscripts, presubscripts and arguments to a symbol.

```
M := script(Mammoth, [ [i,j],[k,l],[0,1],[2],[u,v,w] ])
```

$${}_{2}^{0,1}Mammoth_{i,j}^{k,l}\,(u,v,w)$$

Type: Symbol

```
scripts M
```

$$[sub = [i,j], sup = [k,l], presup = [0,1], presub = [2], args = [u,v,w]]$$

Type: Record(sub: List OutputForm, sup: List OutputForm, presup: List OutputForm, presub: List OutputForm, args: List OutputForm)

If trailing lists of scripts are omitted, they are assumed to be empty.

```
N := script(Nut, [ [i,j],[k,l],[0,1] ])
```

$${}^{0,1}Nut_{i,j}^{k,l}$$

Type: Symbol

```
scripts N
```

$$[sub = [i,j], sup = [k,l], presup = [0,1], presub = [], args = []]$$

Type: Record(sub: List OutputForm, sup: List OutputForm, presup: List OutputForm, presub: List OutputForm, args: List OutputForm)

9.64 Table

The `Table` constructor provides a general structure for associative storage. This type provides hash tables in which data objects can be saved according to keys of any type. For a given table, specific types must be chosen for the keys and entries.

In this example the keys to the table are polynomials with integer coefficients. The entries in the table are strings.

```
t:  Table(Polynomial Integer, String) := table()
```

$$table()$$

Type: Table(Polynomial Integer,String)

To save an entry in the table, the **setelt** operation is used. This can be called directly, giving the table a key and an entry.

```
setelt(t, x**2 - 1, "Easy to factor")
```

```
"Easy to factor"
```

Type: String

Alternatively, you can use assignment syntax.

```
t(x**3 + 1) := "Harder to factor"
```

```
"Harder to factor"
```

Type: String

```
t(x) := "The easiest to factor"
```

```
"The easiest to factor"
```

Type: String

Entries are retrieved from the table by calling the **elt** operation.

```
elt(t, x)
```

```
"The easiest to factor"
```

Type: String

This operation is called when a table is "applied" to a key using this or the following syntax.

```
t.x
```

```
"The easiest to factor"
```

Type: String

```
t x
```

```
"The easiest to factor"
```

Type: String

Parentheses are used only for grouping. They are needed if the key is an infixed expression.

```
t.(x**2 - 1)
```

```
"Easy to factor"
```

Type: String

Note that the **elt** operation is used only when the key is known to be in the table—otherwise an error is generated.

```
t (x**3 + 1)
```

"Harder to factor"

Type: String

You can get a list of all the keys to a table using the **keys** operation.

```
keys t
```

$$\left[x, x^3 + 1, x^2 - 1\right]$$

Type: List Polynomial Integer

If you wish to test whether a key is in a table, the **search** operation is used. This operation returns either an entry or `"failed"`.

```
search(x, t)
```

"The easiest to factor"

Type: Union(String,...)

```
search(x**2, t)
```

"failed"

Type: Union("failed",...)

The return type is a union so the success of the search can be tested using `case`.

```
search(x**2, t) case "failed"
```

true

Type: Boolean

The **remove** operation is used to delete values from a table.

```
remove!(x**2-1, t)
```

"Easy to factor"

Type: Union(String,...)

If an entry exists under the key, then it is returned. Otherwise **remove** returns `"failed"`.

```
remove!(x-1, t)
```

"failed"

Type: Union("failed",...)

The number of key-entry pairs can be found using the # operation.

```
#t
```

2

Type: PositiveInteger

Just as **keys** returns a list of keys to the table, a list of all the entries can be obtained using the **members** operation.

```
members t
```

["The easiest to factor","Harder to factor"]

Type: List String

A number of useful operations take functions and map them on to the table to compute the result. Here we count the entries which have "`Hard`" as a prefix.

```
count(s: String +-> prefix?("Hard", s), t)
```

1

Type: PositiveInteger

Other table types are provided to support various needs.

> `AssociationList` gives a list with a table view. This allows new entries to be appended onto the front of the list to cover up old entries. This is useful when table entries need to be stacked or when frequent list traversals are required. See 9.1 on page 1 for more information.
>
> `EqTable` gives tables in which keys are considered equal only when they are in fact the same instance of a structure. See 9.16 on page 61 for more information.
>
> `StringTable` should be used when the keys are known to be strings. See 9.62 on page 210 for more information.

> SparseTable provides tables with default entries, so lookup never fails. The GeneralSparseTable constructor can be used to make any table type behave this way. See 9.58 on page 200 for more information.

> KeyedAccessFile allows values to be saved in a file, accessed as a table. See 9.33 on page 112 for more information.

Issue the system command `)show Table` to display the full list of operations defined by `Table`.

9.65 TextFile

The domain `TextFile` allows AXIOM to read and write character data and exchange text with other programs. This type behaves in AXIOM much like a `File` of strings, with additional operations to cause new lines. We give an example of how to produce an upper case copy of a file.

This is the file from which we read the text.

```
f1: TextFile := open("/etc/motd", "input")

                              "/etc/motd"

                                                     Type: TextFile
```

This is the file to which we write the text.

```
f2: TextFile := open("/tmp/MOTD", "output")

                              "/tmp/MOTD"

                                                     Type: TextFile
```

Entire lines are handled using the **readLine** and **writeLine** operations.

```
l := readLine! f1

              "Risc System/6000 Model 320H: pascal"

                                                       Type: String
```

```
writeLine!(f2, upperCase l)

              "RISC SYSTEM/6000 MODEL 320H: PASCAL"

                                                       Type: String
```

Use the **endOfFile?** operation to check if you have reached the end of the file.

```
while not endOfFile? f1 repeat
    s := readLine! f1
    writeLine!(f2, upperCase s)
```

Type: Void

The file `f1` is exhausted and should be closed.

```
close! f1
```

"/etc/motd"

Type: TextFile

It is sometimes useful to write lines a bit at a time. The **write** operation allows this.

```
write!(f2, "-The-")
```

"-The-"

Type: String

```
write!(f2, "-End-")
```

"-End-"

Type: String

This ends the line. This is done in a machine-dependent manner.

```
writeLine! f2
```

""

Type: String

```
close! f2
```

"/tmp/MOTD"

Type: TextFile

Finally, clean up.

```
)system rm /tmp/MOTD
```

For more information on related topics, see 9.21 on page 74, 9.33 on page 112, and 9.34 on page 116. Issue the system command `)show TextFile` to display the full list of operations defined by `TextFile`.

9.66 TwoDimensionalArray

The `TwoDimensionalArray` domain is used for storing data in a two dimensional data structure indexed by row and by column. Such an array is a homogeneous data structure in that all the entries of the array must belong to the same AXIOM domain. Each array has a fixed number of rows and columns specified by the user and arrays are not extensible. In AXIOM, the indexing of two-dimensional arrays is one-based. This means that both the "first" row of an array and the "first" column of an array are given the index `1`. Thus, the entry in the upper left corner of an array is in position `(1,1)`.

The operation **new** creates an array with a specified number of rows and columns and fills the components of that array with a specified entry. The arguments of this operation specify the number of rows, the number of columns, and the entry.

This creates a five-by-four array of integers, all of whose entries are zero.

```
arr :  ARRAY2 INT := new(5,4,0)
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

`Type:  TwoDimensionalArray Integer`

The entries of this array can be set to other integers using the operation **setelt**.

Issue this to set the element in the upper left corner of this array to `17`.

```
setelt(arr,1,1,17)
```

$$17$$

`Type:  PositiveInteger`

Now the first element of the array is `17.`

```
arr
```

$$\begin{bmatrix} 17 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

`Type:  TwoDimensionalArray Integer`

Likewise, elements of an array are extracted using the operation **elt**.

```
elt(arr,1,1)
```

$$17$$

Type: PositiveInteger

Another way to use these two operations is as follows. This sets the element in position `(3,2)` of the array to `15`.

```
arr(3,2) := 15
```

$$15$$

Type: PositiveInteger

This extracts the element in position `(3,2)` of the array.

```
arr(3,2)
```

$$15$$

Type: PositiveInteger

The operations **elt** and **setelt** come equipped with an error check which verifies that the indices are in the proper ranges. For example, the above array has five rows and four columns, so if you ask for the entry in position `(6,2)` with `arr(6,2)` AXIOM displays an error message. If there is no need for an error check, you can call the operations **qelt** and **qsetelt** which provide the same functionality but without the error check. Typically, these operations are called in well-tested programs.

The operations **row** and **column** extract rows and columns, respectively, and return objects of `OneDimensionalArray` with the same underlying element type.

```
row(arr,1)
```

$$[17, 0, 0, 0]$$

Type: OneDimensionalArray Integer

```
column(arr,1)
```

$$[17, 0, 0, 0, 0]$$

Type: OneDimensionalArray Integer

You can determine the dimensions of an array by calling the operations **nrows** and **ncols**, which return the number of rows and columns, respectively.

```
nrows(arr)
```

$$5$$

Type: PositiveInteger

```
ncols(arr)
```

$$4$$

Type: PositiveInteger

To apply an operation to every element of an array, use **map**. This creates a new array. This expression negates every element.

```
map(-,arr)
```

$$\begin{bmatrix} -17 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -15 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Type: TwoDimensionalArray Integer

This creates an array where all the elements are doubled.

```
map((x +-> x + x),arr)
```

$$\begin{bmatrix} 34 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 30 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Type: TwoDimensionalArray Integer

To change the array destructively, use **map** instead of **map**. If you need to make a copy of any array, use **copy**.

```
arrc := copy(arr)
```

$$\begin{bmatrix} 17 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 15 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Type: TwoDimensionalArray Integer

```
map!(-,arrc)
```

$$\begin{bmatrix} -17 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -15 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Type: TwoDimensionalArray Integer

```
arrc
```

$$\begin{bmatrix} -17 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -15 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Type: TwoDimensionalArray Integer

```
arr
```

$$\begin{bmatrix} 17 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 15 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Type: TwoDimensionalArray Integer

Use **member?** to see if a given element is in an array.

```
member?(17,arr)
```

true

Type: Boolean

```
member?(10317,arr)
```

false

Type: Boolean

To see how many times an element appears in an array, use **count**.

```
count(17,arr)
```

$$1$$

Type: PositiveInteger

```
count(0,arr)
```

$$18$$

Type: PositiveInteger

For more information about the operations available for `TwoDimensionalArray`, issue `)show TwoDimensionalArray`. For information on related topics, see 9.39 on page 142 and 9.44 on page 158.

9.67 UnivariatePolynomial

The domain constructor `UnivariatePolynomial` (abbreviated `UP`) creates domains of univariate polynomials in a specified variable. For example, the domain `UP(a1,POLY FRAC INT)` provides polynomials in the single variable `a1` whose coefficients are general polynomials with rational number coefficients.

Restriction:

> AXIOM does not allow you to create types where `UnivariatePolynomial` is contained in the coefficient type of `Polynomial`. Therefore, `UP(x,POLY INT)` is legal but `POLY UP (x,INT)` is not.

.

`UP(x,INT)` is the domain of polynomials in the single variable `x` with integer coefficients.

```
(p,q) : UP(x,INT)
```

Type: Void

```
p := (3*x-1)**2 * (2*x + 8)
```

$$18\ x^3+60\ x^2-46\ x+8$$

Type: UnivariatePolynomial(x,Integer)

```
q := (1 - 6*x + 9*x**2)**2
```

$$81\ x^4-108\ x^3+54\ x^2-12\ x+1$$

Type: UnivariatePolynomial(x,Integer)

The usual arithmetic operations are available for univariate polynomials.

```
p**2 + p*q
```

$$1458\ x^7 + 3240\ x^6 - 7074\ x^5 + 10584\ x^4 - 9282\ x^3 + 4120\ x^2 - 878\ x + 72$$

Type: UnivariatePolynomial(x,Integer)

The operation **leadingCoefficient** extracts the coefficient of the term of highest degree.

```
leadingCoefficient p
```

$$18$$

Type: PositiveInteger

The operation **degree** returns the degree of the polynomial. Since the polynomial has only one variable, the variable is not supplied to operations like **degree**.

```
degree p
```

$$3$$

Type: PositiveInteger

The reductum of the polynomial, the polynomial obtained by subtracting the term of highest order, is returned by **reductum**.

```
reductum p
```

$$60\ x^2 - 46\ x + 8$$

Type: UnivariatePolynomial(x,Integer)

The operation **gcd** computes the greatest common divisor of two polynomials.

```
gcd(p,q)
```

$$9\ x^2 - 6\ x + 1$$

Type: UnivariatePolynomial(x,Integer)

The operation **lcm** computes the least common multiple.

```
lcm(p,q)
```

$$162\ x^5 + 432\ x^4 - 756\ x^3 + 408\ x^2 - 94\ x + 8$$

Type: UnivariatePolynomial(x,Integer)

The operation **resultant** computes the resultant of two univariate polynomials. In the case of `p` and `q`, the resultant is `0` because they share a common root.

```
resultant(p,q)
```

$$0$$

Type: NonNegativeInteger

To compute the derivative of a univariate polynomial with respect to its variable, use **D**.

```
D p
```

$$54\ x^2 + 120\ x - 46$$

Type: UnivariatePolynomial(x,Integer)

Univariate polynomials can also be used as if they were functions. To evaluate a univariate polynomial at some point, apply the polynomial to the point.

```
p(2)
```

$$300$$

Type: PositiveInteger

The same syntax is used for composing two univariate polynomials, i.e. substituting one polynomial for the variable in another. This substitutes `q` for the variable in `p`.

```
p(q)
```

$$9565938\ x^{12} - 38263752\ x^{11} + 70150212\ x^{10} - 77944680\ x^9 + 58852170\ x^8 -$$

$$32227632\ x^7 + 13349448\ x^6 - 4280688\ x^5 + 1058184\ x^4 -$$

$$192672\ x^3 + 23328\ x^2 - 1536\ x + 40$$

Type: UnivariatePolynomial(x,Integer)

This substitutes `p` for the variable in `q`.

```
q(p)
```

$$8503056\ x^{12} + 113374080\ x^{11} + 479950272\ x^{10} + 404997408\ x^9 -$$
$$1369516896\ x^8 - 626146848\ x^7 + 2939858712\ x^6 - 2780728704\ x^5 +$$
$$1364312160\ x^4 - 396838872\ x^3 + 69205896\ x^2 - 6716184\ x + 279841$$

Type: UnivariatePolynomial(x,Integer)

To obtain a list of coefficients of the polynomial, use **coefficients**.

```
l := coefficients p
```

$$[18, 60, -46, 8]$$

Type: List Integer

From this you can use **gcd** and **reduce** to compute the content of the polynomial.

```
reduce(gcd,l)
```

$$2$$

Type: PositiveInteger

Alternatively (and more easily), you can just call **content**.

```
content p
```

$$2$$

Type: PositiveInteger

Note that the operation **coefficients** omits the zero coefficients from the list. Sometimes it is useful to convert a univariate polynomial to a vector whose i-th position contains the degree `i-1` coefficient of the polynomial.

```
ux := (x**4+2*x+3)::UP(x,INT)
```

$$x^4 + 2\ x + 3$$

Type: UnivariatePolynomial(x,Integer)

To get a complete vector of coefficients, use the operation **vectorise**, which takes a univariate polynomial and an integer denoting the length of the desired vector.

```
vectorise(ux,5)
```

$$[3, 2, 0, 0, 1]$$

Type: Vector Integer

It is common to want to do something to every term of a polynomial, creating a new polynomial in the process.

This is a function for iterating across the terms of a polynomial, squaring each term.

```
squareTerms(p) == reduce(+,[t**2 for t in monomials p])
```

Type: Void

Recall what `p` looked like.

```
p
```

$$18\ x^3 + 60\ x^2 - 46\ x + 8$$

Type: UnivariatePolynomial(x,Integer)

We can demonstrate `squareTerms` on `p`.

```
squareTerms p
```

```
Compiling function squareTerms with type
  UnivariatePolynomial(x,Integer) ->
    UnivariatePolynomial(x,Integer)
```

$$324\ x^6 + 3600\ x^4 + 2116\ x^2 + 64$$

Type: UnivariatePolynomial(x,Integer)

When the coefficients of the univariate polynomial belong to a field,[7] it is possible to compute quotients and remainders.

```
(r,s) :  UP(a1,FRAC INT)
```

Type: Void

```
r := a1**2 - 2/3
```

[7]For example, when the coefficients are rational numbers, as opposed to integers. The important property of a field is that non-zero elements can be divided and produce another element. The quotient of the integers 2 and 3 is not another integer.

$$a1^2 - \frac{2}{3}$$

Type: UnivariatePolynomial(a1,Fraction Integer)

```
s := a1 + 4
```

$$a1 + 4$$

Type: UnivariatePolynomial(a1,Fraction Integer)

When the coefficients are rational numbers or rational expressions, the operation **quo** computes the quotient of two polynomials.

```
r quo s
```

$$a1 - 4$$

Type: UnivariatePolynomial(a1,Fraction Integer)

The operation **rem** computes the remainder.

```
r rem s
```

$$\frac{46}{3}$$

Type: UnivariatePolynomial(a1,Fraction Integer)

The operation **divide** can be used to return a record of both components.

```
d := divide(r, s)
```

$$\left[quotient = a1 - 4, remainder = \frac{46}{3}\right]$$

Type: Record(quotient: UnivariatePolynomial(a1,Fraction Integer), remainder: UnivariatePolynomial(a1,Fraction Integer))

Now we check the arithmetic!

```
r - (d.quotient * s + d.remainder)
```

$$0$$

Type: UnivariatePolynomial(a1,Fraction Integer)

It is also possible to integrate univariate polynomials when the coefficients belong to a field.

```
integrate r
```

$$\frac{1}{3}\ a1^3 - \frac{2}{3}\ a1$$

Type: UnivariatePolynomial(a1,Fraction Integer)

```
integrate s
```

$$\frac{1}{2}\ a1^2 + 4\ a1$$

Type: UnivariatePolynomial(a1,Fraction Integer)

One application of univariate polynomials is to see expressions in terms of a specific variable.

We start with a polynomial in `a1` whose coefficients are quotients of polynomials in `b1` and `b2`.

```
t : UP(a1,FRAC POLY INT)
```

Type: Void

Since in this case we are not talking about using multivariate polynomials in only two variables, we use `Polynomial`. We also use `Fraction` because we want fractions.

```
t := a1**2 - a1/b2 + (b1**2-b1)/(b2+3)
```

$$a1^2 - \frac{1}{b2}\ a1 + \frac{b1^2 - b1}{b2 + 3}$$

Type: UnivariatePolynomial(a1,Fraction Polynomial Integer)

We push all the variables into a single quotient of polynomials.

```
u : FRAC POLY INT := t
```

$$\frac{a1^2\ b2^2 + \left(b1^2 - b1 + 3\ a1^2 - a1\right)\ b2 - 3\ a1}{b2^2 + 3\ b2}$$

Type: Fraction Polynomial Integer

Alternatively, we can view this as a polynomial in the variable This is a *mode-directed* conversion: you indicate as much of the structure as you care about and let AXIOM decide on the full type and how to do the transformation.

```
u :: UP(b1,?)
```

$$\frac{1}{b2 + 3}\ b1^2 - \frac{1}{b2 + 3}\ b1 + \frac{a1^2\ b2 - a1}{b2}$$

Type: UnivariatePolynomial(b1,Fraction Polynomial Integer)

For more information on related topics, see 9.49 on page 174, 9.41 on page 153, and 9.15 on page 59. Issue the system command `)show UnivariatePolynomial` to display the full list of operations defined by `UnivariatePolynomial`.

9.68 UniversalSegment

The `UniversalSegment` domain generalizes `Segment` by allowing segments without a "hi" end point.

```
pints := 1..
```

$$1..$$

Type: UniversalSegment PositiveInteger

```
nevens := (0..) by -2
```

$$0..\mathrm{by}\ -2$$

Type: UniversalSegment NonNegativeInteger

Values of type `Segment` are automatically converted to type `UniversalSegment` when appropriate.

```
useg: UniversalSegment(Integer) := 3..10
```

$$3..10$$

Type: UniversalSegment Integer

The operation **hasHi** is used to test whether a segment has a `hi` end point.

```
hasHi pints
```

false

Type: Boolean

```
hasHi nevens
```

false

Type: Boolean

```
hasHi useg
```

true

Type: Boolean

All operations available on type `Segment` apply to `UniversalSegment`, with the proviso that expansions produce streams rather than lists. This is to accommodate infinite expansions.

```
expand pints
```

$$[1, 2, 3, 4, 5, 6, 7, \ldots]$$

Type: `Stream Integer`

```
expand nevens
```

$$[0, -2, -4, -6, -8, -10, -12, \ldots]$$

Type: `Stream Integer`

```
expand [1, 3, 10..15, 100..]
```

$$[1, 3, 10, 11, 12, 13, 14, \ldots]$$

Type: `Stream Integer`

For more information on related topics, see 9.53 on page 189, 9.54 on page 191, 9.36 on page 129, and 9.60 on page 202. Issue the system command `)show UniversalSegment` to display the full list of operations defined by `UniversalSegment`.

9.69 Vector

The `Vector` domain is used for storing data in a one-dimensional indexed data structure. A vector is a homogeneous data structure in that all the components of the vector must belong to the same AXIOM domain. Each vector has a fixed length specified by the user; vectors are not extensible. This domain is similar to the `OneDimensionalArray` domain, except that when the components of a `Vector` belong to a `Ring`, arithmetic operations are provided. For more examples of operations that are defined for both `Vector` and `OneDimensionalArray`, see 9.44 on page 158.

As with the `OneDimensionalArray` domain, a `Vector` can be created by calling the operation **new**, its components can be accessed by calling the operations **elt** and **qelt**, and its components can be reset by calling the operations **setelt** and **qsetelt**.

This creates a vector of integers of length `5` all of whose components are `12`.

```
u : VECTOR INT := new(5,12)
```

$$[12, 12, 12, 12, 12]$$

Type: `Vector Integer`

This is how you create a vector from a list of its components.

```
v : VECTOR INT := vector([1,2,3,4,5])
```

$$[1, 2, 3, 4, 5]$$

Type: Vector Integer

Indexing for vectors begins at 1. The last element has index equal to the length of the vector, which is computed by "#".

```
#(v)
```

$$5$$

Type: PositiveInteger

This is the standard way to use **elt** to extract an element. Functionally, it is the same as if you had typed `elt(v,2)`.

```
v.2
```

$$2$$

Type: PositiveInteger

This is the standard way to use **setelt** to change an element. It is the same as if you had typed `setelt(v,3,99)`.

```
v.3 := 99
```

$$99$$

Type: PositiveInteger

Now look at `v` to see the change. You can use **qelt** and **qsetelt** (instead of **elt** and **setelt**, respectively) but *only* when you know that the index is within the valid range.

```
v
```

$$[1, 2, 99, 4, 5]$$

Type: Vector Integer

When the components belong to a `Ring`, AXIOM provides arithmetic operations for `Vector`. These include left and right scalar multiplication.

```
5 * v
```

$$[5, 10, 495, 20, 25]$$

Type: Vector Integer

```
v * 7
```

$$[7, 14, 693, 28, 35]$$

Type: Vector Integer

```
w : VECTOR INT := vector([2,3,4,5,6])
```

$$[2, 3, 4, 5, 6]$$

Type: Vector Integer

Addition and subtraction are also available.

```
v + w
```

$$[3, 5, 103, 9, 11]$$

Type: Vector Integer

Of course, when adding or subtracting, the two vectors must have the same length or an error message is displayed.

```
v - w
```

$$[-1, -1, 95, -1, -1]$$

Type: Vector Integer

For more information about other aggregate domains, see the following: 9.36 on page 129, 9.39 on page 142, 9.44 on page 158, 9.55 on page 192, 9.64 on page 215, and 9.66 on page 221. Issue the system command `)show Vector` to display the full list of operations defined by `Vector`.

9.70 Void

When an expression is not in a value context, it is given type `Void`. For example, in the expression

```
r := (a; b; if c then d else e; f)
```

values are used only from the subexpressions `c` and `f`: all others are thrown away. The subexpressions `a`, `b`, `d` and `e` are evaluated for side-effects only and have type `Void`. There is a unique value of type `Void`.

You will most often see results of type `Void` when you declare a variable.

```
a : Integer
```

```
                                                        Void
```

Usually no output is displayed for `Void` results. You can force the display of a rather ugly object by issuing `)set message void on`.

```
)set message void on
```

```
b : Fraction Integer
```

```
                              "()"
                                                  Type: Void
```

```
)set message void off
```

All values can be converted to type `Void`.

```
3::Void
```

```
                                                  Type: Void
```

Once a value has been converted to `Void`, it cannot be recovered.

```
% :: PositiveInteger
```

```
Cannot convert from type Void to PositiveInteger for value "()"
```