Diesel Standard Library Reference Manual

The Cecil Group

Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle, Washington 98195-2350
(206) 685-2094; fax: (206) 543-2969
cecil@cs.washington.edu

Vortex 3.3
February, 2006

Contents

1 Introduction

The Diesel standard library defines a collection of data structures and control structures used by most Diesel programs. Since even basic structures that would be built into other languages — like integers, conditional statements, and loops — are defined at user level, it is easy to define and use new control structures and new operations on the standard data types.

This manual describes the data and control structures that make up the `noeval` standard library. They are available to a program that is compiled by the Vortex compiler with the `stdlib` (default) or `noevalstdlib` standard library levels, or that explicitly includes the file `prelude.noeval.diesel`. The "How To Use Vortex" document describes the additional functionality of the evaluator which is available to each program compiled at the `stdlib` level. A minimal standard library (`smallstdlib`) is also available.

This manual presents data and control structures in five sections:

- basic data types - including "simple" data types such as void, int, float, char, and pair
- basic control structures - including bool and closure data types and if and while control structures
- collections - includes arrays, sets, hash tables, strings, lists, etc.
- streams and I/O - includes streams and file-based I/O operations
- miscellaneous - includes some system operations and some other data types and operations

with the last section describing precedences of binary operators defined in the library.

Verbal description is generally preceded by Diesel code declaring the classes and functions. Function bodies are not shown; the headers show "types" of arguments and the result. Overriding methods are not shown unless they also introduced refined signatures. Examples sometimes follow the verbal description. Finally, file names refer the user to the source code for the most complete and up-to-date details.

When appropriate, abstract interfaces are described in the first part of a subsection and concrete implementations of these interfaces in the second part.

1.1 Tips

This section may be skipped on first reading.

Some names used in the standard library differ from those found in the traditional languages like C. In particular, fixed-length indexed arrays are called "vectors." The name "array" is used for variable-length indexed arrays which allow adding and removing elements from both ends and can expand or shrink at runtime depending on the number of elements they are holding. The indexing operation is denoted by "`!`" (which is an infix version of `fetch`), because the traditional bracket notation "[]" is used by the Diesel language to indicate parameterization of classes/types and functions. For example, printing out an array element (or, more generally, a table element) can be done like this: `(a!i).print_line;`.

Creation of new instances of classes at runtime is typically achieved by `new_` methods which encapsulate the `new` object creation and initialization expressions. The name of such a method usually starts with `new_` followed by the name of the concrete class whose instance is being created; sometimes a suffix is used to distinguish between different versions. However, this is merely a convention. Notable exceptions from this convention are methods that construct `cons` cells, `pair`s, `triple`s, `quadruple`s, and `quintuple`s, which don't start with `new_`. Here are some examples:

```
-- create an uninitialized m_vector
fun new_m_vector[T](size:int):m_vector[T];

-- initialize the created m_vector with filler
fun new_m_vector[T](size:int, filler:T):m_vector[T];

-- use the init closure to compute the initial values
fun new_m_vector_init[T](size:int, init:&(int):T):m_vector[T];
```

Many operations that may produce an error or a special condition (e.g., indexing an array) take an optional last argument which is the closure to be invoked in such case. Thus, a programmer may either ignore the possibility of an error (which would halt the program) or handle the error condition (e.g., return a certain object when the array index is out of bounds). The default value of the closure typically invokes the `error` method, which prints a message, offers a debugging prompt, and then terminates the program. For example:

```
-- call error if the key is not found
fun fetch(t:table[‘Key,‘Value], key:Key):Value;

-- invoke if_absent if the key is not found
fun fetch(t:table[‘Key,‘Value], key:Key, if_absent:&():‘Else):Value|Else;
```

2 Basic data types and operations

2.1 Maximal and minimal types, identity testing, and printing

In `general.diesel`:

The following types, classes, and/or objects are special, and predefined:

`void` is the type of methods that do not return a "real" result. `void` is implicitly a supertype of all other types. The `void` object can be returned explicitly from a method, if necessary.

```
predefined object void;
```

`any` is implicitly the supertype of all other non-`void` types. It can be used as the argument type of functions that work over any (non-void) object.

```
predefined abstract class any;
```

The `none` type is the result type of functions that never return to their callers, and the type of closure arguments that are never invoked. `none` is a subtype of all other types, and there is no object of this type.

```
predefined type none;
```

The `dynamic` type disables static type checking. It's used to explicitly declare the type of a variable that cannot be statically checked. A value of any type can be assigned to a variable of type dynamic, and vice versa. (The Diesel compiler will perform run-time typechecking to ensure type safety even in the face of uses of dynamic or of static type errors.)

```
predefined type dynamic;

extend module Stdlib;

module General;
```

All objects can be compared for identity. Two things that print out the same may not be `==`, e.g. `"abc" == "abc"` may return `false`. By default, operations on collections that compare elements, such as finding an element in a list or adding an element to a set, use `=` (implemented by `comparable` objects) rather than `==`, unless the collection's name includes `identity_`.

```
fun ==(l:any, r:any):bool;  -- object identity test (use = instead for value equality testing)
fun !==(l:any, r:any):bool;  -- not ==
```

Any object can be converted to a string, although the default version can be low-level and ugly. Most commonly-used objects override `print_string` to return something prettier. (A two-element version of `print` permits strings to be printed to `files`.)

```
fun print_string(x:any):string;  -- return a string print version
fun print(x:any):void;  -- print the print_string
fun print_line(x:any):void;  -- print the print_string and a newline
fun print_line():void;  -- just print a newline
```

The `error` method is the standard way to prematurely quit execution of a Diesel program.

```
fun error(msg:ordered_collection[`T]):none;  -- quits with an error message; does not return
```

`upcast[T]` changes the static type of its argument to T from a

```
fun upcast[T](x:`S <= T):T;

end module General;

end module Stdlib;
```

2.2 Equality, ordering, and hashing

In `comparable.diesel`:

```
extend module Stdlib;
```

`comparable` objects can be tested for equality. This equality should be an abstract notion of equality which compares two abstract values, as appropriate to the kind of data types being compared. It is not an object identity test such as `==`, which, like Lisp's `eq`, tests "pointer equality." By default, operations that compare elements use `=` rather than `==`, unless the operation's name includes `identity_`. Subclasses should implement `=`; `!=` is derived from it.

```
module Comparable;
  abstract class comparable[T <= comparable[T]] isa comparable[‘S <= T];
  fun =(x1:‘T <= comparable[T], x2:‘T <= comparable[T]):bool;  -- equal (abstract) values?
  fun !=(x1:‘T <= comparable[T], x2:‘T <= comparable[T]):bool;  -- not =
```

`identity_comparable` objects are `comparable` objects that implement `=` in terms of `==` by default.

```
module IdentityComparable;
  abstract class identity_comparable[T <= identity_comparable[T]]
        isa comparable[T], identity_comparable[‘S <= T];
```

`partially_ordered` objects are `comparable` objects that also can be compared for being less than one another, and other combinations. Since they form only a partial order, `not(a < b)` does not imply `a >= b`. The `compare` function implements a four-way comparison, invoking one of its argument closures depending on how the first argument compares to its second argument. By default, subclasses should implement `<`; the other operations are defined in terms of `<` and `=`. Alternatively, subclasses can inherit instead from `partially_ordered_using_<=` and then implement `<=` instead of `<`.

```
module PartiallyOrdered;
  abstract class partially_ordered[T <= partially_ordered[T]]
        isa comparable[T], partially_ordered[‘S <= T];
  fun < (x1:‘T <= partially_ordered[T],
              x2:‘T <= partially_ordered[T]):bool;  -- x1 ordered below x2?
  fun <=(x1:‘T <= partially_ordered[T],
              x2:‘T <= partially_ordered[T]):bool;  -- < or =
  fun > (x1:‘T <= partially_ordered[T],
              x2:‘T <= partially_ordered[T]):bool;  -- x2 < x1?
  fun >=(x1:‘T <= partially_ordered[T],
              x2:‘T <= partially_ordered[T]):bool;  -- > or =
```

The `PartiallyOrderedCompareResult` abstract class and its four concrete objects, `Equal`, `LessThan`, `GreaterThan`, and `Unordered`, represent the possible outcomes of comparing two `partially_ordered` values.

```
  abstract class PartiallyOrderedCompareResult
                      isa identity_comparable[PartiallyOrderedCompareResult];
    abstract class OrderedCompareResult
                                      isa PartiallyOrderedCompareResult;
      object Equal isa OrderedCompareResult;
      object LessThan isa OrderedCompareResult;
      object GreaterThan isa OrderedCompareResult;
    object Unordered isa PartiallyOrderedCompareResult;
  fun compare(x1:‘T <= partially_ordered[T],
                    x2:‘T <= partially_ordered[T],
                    if_less:&():‘S, if_equal:&():‘S, if_greater:&():‘S,
                    if_unordered:&():‘S):S;
```

```
  fun compare(x1:‘T <= partially_ordered[T],
                    x2:‘T <= partially_ordered[T]
                    ):PartiallyOrderedCompareResult;  -- compare two partially-ordered values, returning an outcome
module PartiallyOrderedUsingLE;
  abstract class partially_ordered_using_<=
                                        [T <= partially_ordered_using_<=[T]]
        isa partially_ordered[T], partially_ordered_using_<=[‘S <= T];
```

ordered objects refine partially_ordered objects by imposing a total order, i.e., not(a < b) does imply a >= b. The min and max functions are defined for all ordered objects. The compare function implements a three-way comparison, invoking one of its argument closures depending on how the first argument compares to its second argument. By default, concrete subclasses provide implementations of = and <, and compare is provided by default in terms of those primitives. Alternatively, a concrete subclass can inherit from ordered_using_compare, which assumes the subclass will provide an implementation of compare and defines = and < in terms of it.

```
module Ordered;
  abstract class ordered[T <= ordered[T]] isa partially_ordered[T],
                                                    ordered[‘S <= T];
  fun min(x1:‘T <= ordered[T], x2:‘T <= ordered[T]):T;
  fun max(x1:‘T <= ordered[T], x2:‘T <= ordered[T]):T;
  fun compare(x1:‘T <= ordered[T], x2:‘T <= ordered[T],
                    if_less:&():‘S, if_equal:&():‘S, if_greater:&():‘S
                    ):S;
module OrderedUsingCompare;  -- compare two ordered values, returning an outcome
  abstract class ordered_using_compare[T <= ordered_using_compare[T]]
            isa ordered[T], ordered_using_compare[‘S <= T];
```

A hashable object is a comparable (equality-testable) object that also supports a hash function. hash(x, r) returns an integer in the range [0..r-1], subject to the constraint that if a = b, then hash(a, r) = hash(b, r).

```
module Hashable;
  abstract class hashable[T <= hashable[T]] isa comparable[T],
                                                      hashable[‘S <= T];
  fun hash(t:hashable[‘T], range:int):int;
```

An identity_hashable object is both hashable and identity_comparable.

```
module IdentityHashable;
  abstract class identity_hashable[T <= identity_hashable[T]]
        isa identity_comparable[T], hashable[T], identity_hashable[‘S <= T];
```

An ordered_hashable object is both hashable and ordered.

```
module OrderedHashable;
  abstract class ordered_hashable[T <= ordered_hashable[T]]
        isa ordered[T], hashable[T], ordered_hashable[‘S <= T];
```

2.3 Numbers

In number.diesel:

Numbers support the standard arithmetic operations, plus the protocol of totally-ordered objects (=, <, min, etc.). This contract implies that all subclasses of number are freely mixable at run-time.

```
extend module Stdlib;

module Number;
abstract class num isa ordered_hashable[num];
```

`as_int` converts its argument to an integer, possibly rounding up or down if not already an integer.

```
fun as_int(:num):integer;
```

`as_float` converts its argument to a float representation. Related operations exist to specify the kind of float to produce.

```
fun as_float(n:num):float;
fun as_single_float(:num):single_float;
fun as_double_float(:num):double_float;
```

Standard arithmetic operations. When invoked on a small integer (an `int`), they might overflow and wrap around. (Use the `_ov` versions below to get correct answers in the face of overflow.)

```
fun +(:‘T <= num, :T):T;
fun -(:‘T <= num, :T):T;
fun *(:‘T <= num, :T):T;
```

`/` returns an integer if applied to integers, rounding towards zero. Use `/_float` to yield a (precise) `float` result.

```
fun /(:‘T <= num, :T):T;
fun /_float(:‘T <= num, :T):T & float;
```

Unary arithmetic operators:

```
fun negate(n:‘T <= num):T;
fun -(n:‘T <= num):T;   -- same as negate
fun +(n:‘T <= num):T;   -- silly no-op
```

These arithmetic operators handle `int` overflow by converting to arbitrary-precision integers (`big_ints`):

```
fun -_ov(l:num):num;
fun +_ov(l:num,r:num):num;
fun -_ov(l:num,r:num):num;
fun *_ov(l:num,r:num):num;
fun /_ov(l:num,r:num):num;
fun /_float_ov(l:num,r:num):num;
```

Simple convenience functions:

```
fun pred(i:‘T <= num):T;   -- i-1
fun succ(i:‘T <= num):T;   -- i+1
fun square(n:‘T <= num):T;   -- n2̂
fun cube(n:‘T <= num):T;   -- n3̂
fun abs(n:‘T <= num):T;   -- absolute value
fun sign(x:num):int;   -- return -1, 0, or 1 depending on sign of argument
fun average(n1:‘T <= num, n2:‘T):T|int;   -- arithmetic average: (n1+n2)/2
fun power(x:‘T <= num, power:integer):S where *_ov(:T,:T):‘S <= T;   -- exponentiation
fun **(x:‘T <= num, power:integer):S where *_ov(:T,:T):‘S <= T;   -- same as power

end module Number;
```

2.3.1 Integers

In `integer.diesel`:

Integers are numbers, supporting all the operations of numbers. The normal `+`, `-`, `*`, and `/` operations do not trap overflow; the `+_ov`, etc. functions transparently coerce to arbitrary-precision integers if overflow happens. The bitwise operations assume two's complement representation.

```
extend module Stdlib;

module Integer;
abstract class integer isa num;
```

Modulo and remainder operations:

```
fun mod(l:integer,r:‘T <= integer):T;  -- modulus
fun %(:integer, :‘T <= integer):T;  -- same as mod
fun rem(:integer, :‘T <= integer):T;  -- remainder
fun %_ov(:integer, :‘T <= integer):T;  -- version of % that handles overflow
```

Bitwise operations:

```
fun <<(l:integer,r:integer):integer;  -- left shift
fun >>(l:integer,r:integer):integer;  -- right arithmetic shift (extends sign)
fun <<_ov(l:integer,r:integer):integer;
fun >>_ov(l:integer,r:integer):integer;
fun bit_and (l:integer, r:integer):integer;
fun bit_or (l:integer, r:integer):integer;
fun bit_xor (l:integer, r:integer):integer;
fun bit_xnor (l:integer, r:integer):integer;
fun bit_not(l:integer):integer;
fun get_bit(i:integer, bit:integer):int;  -- extract ith bit (0 or 1)
fun set_bit(i:integer, bit:integer):integer;  -- set ith bit to 1
fun clear_bit(i:integer, bit:integer):integer;  -- set ith bit to 0
```

Conversion operations:

```
fun as_small_int(l:integer):int;
fun as_small_int(:integer, if_overflow:&():‘T):int|T;
```

`as_small_int_if_possible` converts its argument to use an `int` representation if its value is in range, otherwise leaves the argument in its original (`big_int`) representation

```
fun as_small_int_if_possible(l:integer):integer;
```

Operations to manipulate integers in the range [-2ˆ63..2ˆ63), i.e., those integers that can be represented in 8 signed bytes.

```
fun is_int8(val:integer):bool;
fun as_int8(val:integer, if_overflow:&():integer):integer;
```

Logarithm operations:

```
fun log_base(x:integer, base:integer):int;  -- compute logarithm to nearest integer; rounds up
fun exact_log_base(x:integer, base:integer, if_not_exact:&():int):int;  -- compute logarithm, but invoke closure if integer result not exact
```

Test parity:

```
fun is_even(i:integer):bool;
fun is_odd(i:integer):bool;
```

`round_up` rounds toward positive infinity, while `round_down` rounds toward negative infinity. Both ignore the sign of their second argument.

```
fun round_up(i:integer, nearest:integer):integer;
fun round_down(i:integer, nearest:integer):integer;
```

Some standard mathematical functions:

```
fun factorial(n:integer):integer;
fun fibonacci(n:integer):integer;
fun fibonacci_recursive(n:integer):integer;
```

`do_digits_increasing` calls its closure on each digit, optionally specifying the base, from least to most significant; the `position` argument indicates the digit's significance and starts at zero.

```
fun do_digits_increasing(i:integer,
                              c:&(digit:int, position:int):void):void;
fun do_digits_increasing_base(i:integer, base:int,
                                   c:&(digit:int, position:int):void):void;
```

Printing behavior:

```
fun print_string_base(i:integer, base:int):string;  -- specify the base in which to print
fun print_string_padded(i:integer, len:int):string;  -- right-justify the number to be at least len wide
```

The `parse_as_int` methods try to convert a string representation of an integer into the integer representation, calling the optional if_error closure if the string doesn't contain an integer.

```
fun parse_as_int(s:string):integer;
fun parse_as_int(s:string, base:int | &():integer):integer;
fun parse_as_int(s:string, base:int, fail:&():integer):integer;
fun parse_as_small_int(s:string):int;
fun parse_as_small_int(s:string, base:int | &():int):int;
fun parse_as_small_int(s:string, base:int, fail:&():int):int;

end module Integer;
```

In `small-int.diesel`:

Small integers (`int`) are the main representation of integers. They have some fixed size (slightly less than the target machine's natural word size, in the current implementation). Arithmetic operations by default are performed modulo this fixed size, so that overflows silently wrap around. (Arbitrary-precision integers, a.k.a. `big_int`s, handle overflow "properly".)

```
extend module Stdlib;

module SmallInteger;
abstract class int isa integer;
```

Small (signed) integers can be viewed as unsigned integers, and then compared using `_unsigned` operations.

```
fun <_unsigned (l:int, r:int):bool;
fun <=_unsigned (l:int, r:int):bool;
fun >_unsigned (l:int, r:int):bool;
fun >=_unsigned (l:int, r:int):bool;
```

These operations take an extra closure argument that determines what the operation should do if there's an overflow.

```
fun negate_ov (l:int, if_overflow:&():‘T):int|T;
fun add_ov (l:int, r:int, if_overflow:&():‘T):int|T;
fun sub_ov (l:int, r:int, if_overflow:&():‘T):int|T;
fun mul_ov (l:int, r:int, if_overflow:&():‘T):int|T;
fun div_ov (l:int, r:int, if_overflow:&():‘T):int|T;
fun mod_ov (l:int, r:int, if_overflow:&():‘T):int|T;
```

Extra bitwise operations which depend on the fixed size nature of a small integer:

```
fun >>_logical (l:int, r:int):int;  -- logical right shift (no sign-extension)
```

`times_do` on integers is a simple kind of for-loop. Expression `n.times_do(&(i:int){...})` invokes the argument closure `n` times, binding `i` to each of the numbers from `0` to `n-1` in turn; it does not return a value.

```
fun times_do(count:int, c:&(int):void):void;
```

Implementation-dependent properties of a small integer:

```
let num_int_bits:int;  -- number of bits in small int representation
let max_int:int;  -- maximum small int
let min_int:int;  -- minimum small int

end module SmallInteger;
```

In `big-int.diesel`:
Big integers are an arbitrary-precision representation of integers.

```
extend module Stdlib;

module BigInt;
class big_int isa integer;
fun as_big_int(:integer):big_int;

end module BigInt;
```

2.3.2 Floating-point numbers

In `float.diesel`:
Floats are floating-point numbers. Float-specific operations include various kinds of rounding and formatting a floating point number with a particular number of digits after the decimal point. Single floats and double floats are two representations of floats.

```
extend module Stdlib;

module Float;
abstract class float isa num;
```

Trigonometric functions:

```
fun sin(:float):float;
fun cos(:float):float;
fun tan(:float):float;
fun asin(:float):float;
fun acos(:float):float;
fun atan(:float):float;
```

Exponentiation and logarithms:

```
fun exp(n:float):float;  -- en̂
fun log(n:float):float;  -- log base e
fun float_log_base(x:float, base:float):float;  -- log to given base
```

Other operations on real numbers:

```
fun sqrt(:float):float;
```

Rounding up or down to integer values:

```
fun round_as_int(:float):int;  -- round either up or down to an int
fun round(:float):float;  -- round to nearest integral value
fun round_towards_zero(:float):float;  -- round to the nearest integral value towards zero
fun ceiling(:float):float;  -- round up to the nearest integral value
fun floor(:float):float;  -- round down to the nearest integral value
```

Additional float-specific printing operations:

```
fun print_string(:float, num_decimal_places:int):string;  -- specify how many decimal places to print
fun print_string_full(:float):string;  -- print all decimal places
```

Additional float testing behavior:

```
fun is_a_NaN(:float):bool;  -- returns whether the argument is a NaN value

end module Float;
```

In `single-float.diesel`:

```
extend module Stdlib;

module SingleFloat;
abstract class single_float isa float;
```

Single-precision float parsing operations:

```
fun parse_as_float(s:string):single_float;
fun parse_as_float(s:string, if_error:&():single_float):single_float;
```

Single-precision float constants:

```
let pi:float;
let min_positive_single_float:float;
let max_single_float:float;
let min_single_float:float;
let single_float_infinity:float;
let single_float_negative_infinity:float;
let single_float_NaN:float;

end module SingleFloat;
```

In `double-float.diesel`:

```
extend module Stdlib;

module DoubleFloat;
class double_float isa float;
```

Double-precision float parsing operations:

```
fun parse_as_double(s:string):double_float;
fun parse_as_double(s:string, if_error:&():double_float):double_float;
```

Double-precision float constants:

```
let min_positive_double_float:float;
let max_double_float:float;
let min_double_float:float;
let double_float_infinity:float;
let double_float_negative_infinity:float;
let double_float_NaN:float;

end module DoubleFloat;
```

2.4 Characters

In `character.diesel`:

Diesel characters are pretty standard. The `parse_char_as_int` methods convert digits (such as `'0'` ... `'9'`) to integer equivalents.

```
extend module Stdlib;

module Character;
```

`character` is the abstract ancestor of all character-like objects. we assume that all characters can be put into a common integer coding sequence and compared based on the corresponding integer codes. this is true at least of ascii chars and unicode chars, which are the only two kinds currently supported.

```
abstract class character isa ordered_hashable[character];
```

`char_code` returns the integer code for the character in the ascii/unicode character set.

```
fun char_code(:character):int;
```

"Downcasters": test whether the character is a specific kind, and/or convert down to that kind, and/or invoke a second closure (or error) if not that kind:

```
fun if_char(c:character, if_yes:&(char):`T, if_no:&():`T):T;
fun if_char(c:character, if_yes:&(char):void):void;
fun as_char(c:character):char;
fun is_char(c:character):bool;

end module Character;

module Char;
```

`char` represents an ASCII character.

```
abstract class char isa character;
```

Convert between characters and ascii codes:

```
fun from_ascii(i:int):char;  -- convert from ascii code to char
fun from_ascii(i:int, if_error:&():char):char;
fun ascii_code(c:char):int;  -- convert from char to ascii code (same as char_code)
```

Character testing behavior:

```
fun is_lower_case(c:char):bool;
fun is_upper_case(c:char):bool;
fun is_letter(c:char):bool;
fun is_digit(c:char):bool;
fun is_digit(c:char, base:int):bool;
fun is_octal_digit(c:char):bool;
fun is_hex_digit(c:char):bool;
fun is_alphanumeric(c:char):bool;
fun is_white(c:char):bool;
fun is_printable(c:char):bool;
```

Character conversion behavior:

```
fun as_string(c:char):string;  -- convert to a string of a single character
fun to_lower_case(c:char):char;
fun to_upper_case(c:char):char;
```

Parsing operations to convert a digit character to the corresponding int:

```
fun parse_char_as_int(c:char):int;
fun parse_char_as_int(c:char, base:int | &():int):int;
fun parse_char_as_int(c:char, base:int, fail:&():int):int;

end module Char;

module Unicode;
```

Representation and operations for UNICODE 2-byte characters.

```
class unicode_char isa character;
```

Convert between characters and unicode codes:

```
  fun from_unicode(code:int):character;  -- convert from unicode code to character

end module Unicode;
```

2.5 Options

In `absent.diesel`:

The `mb[T]` parameterized type represents either a value of type T or the value `absent`, akin to ML's `option` datatype. `mb[T]` differs from `maybe[T]` in that `mb[T]` doesn't introduce a level of indirection, but cannot store `absent` as a non-absent value. (The current Diesel typechecker also has some limitations in its handling of or-types like `mb[T]`, such as that such a type cannot be directly declared to be subtypes of `comparable[mb[T]]`.)

```
extend module Stdlib;

module Absent;
```

The distinguished `absent` value:

```
object absent;
```

`mb[T]` is either `absent` or a `T`:

```
synonym mb[T] = T | absent;
```

Testers:

```
fun is_present(:mb[‘T]):bool;
fun is_absent(:mb[‘T]):bool;
```

“Typecase” operations:

```
fun if_present(:mb[‘T], if_present:&(T):‘R, if_absent:&():‘R):R;
fun if_present(:mb[‘T], if_present:&(T):void):void;
fun if_absent (:mb[‘T], if_absent:&():‘R, if_present:&(T):‘R):R;
fun if_absent (:mb[‘T], if_absent:&():void):void;
```

Checked “downcast” operations:

```
fun as_present(:mb[‘T]):T;
fun as_present(:mb[‘T], :&():T):T;

end module Absent;
```

2.6 Tuples

In `pair.diesel`:

The `pair`, `triple`, `quadruple`, and `quintuple` data structures represent immutable tuples of values. They can be compared iff their components can be.

```
extend module Stdlib;
```

A `pair` is an immutable pair of arbitrary values. A `pair` of two types `T1` and `T2` is a subtype of pairs of any supertypes `S1` and `S2`, e.g., a `pair` of an `integer` and a `set` is a subtype of a `pair` of a `number` and an `unordered_collection`. The `pair` method constructs a new pair. (The constructor methods for pairs et al. do not start with `new_` – like `cons`, they are exceptions from our naming convention for constructor methods.)

```
module Pair;
class pair[T1,T2] isa pair[‘S1 >= T1, ‘S2 >= T2];
  field first (:pair[‘T1,‘T2]):T1;
  field second(:pair[‘T1,‘T2]):T2;
  fun pair(x:‘T1, y:‘T2):pair[T1,T2];  -- create a pair
extend class pair[‘T1 <= comparable[T1], ‘T2 <= comparable[T2]]
                                                isa comparable[pair[T1,T2]];
extend class pair[‘T1 <= partially_ordered[T1],
                         ‘T2 <= partially_ordered[T2]]
                                isa partially_ordered_using_<=[pair[T1,T2]];
extend class pair[‘T1 <= hashable[T1], ‘T2 <= hashable[T2]]
                                isa hashable[pair[T1,T2]];
fun <=_lex(p1:pair[‘T11 <= ‘T1, ‘T12 <= ‘T2],
                  p2:pair[‘T21 <= ‘T1, ‘T22 <= ‘T2]):bool
                                where ‘T1 <= ordered[T1], ‘T2 <= ordered[T2];  -- lexicographical order


end module Pair;
```

Triples are an immutable triple of arbitrary values. A `triple` of three types `T1`, `T2`, and `T3` is a subtype of triples of any supertypes `S1`, `S2`, and `S3`. The `triple` method constructs a new triple.

```
module Triple;
class triple[T1,T2,T3] isa triple[`S1 >= T1, `S2 >= T2, `S3 >= T3];
  field first (:triple[`T1,`T2,`T3]):T1;
  field second(:triple[`T1,`T2,`T3]):T2;
  field third (:triple[`T1,`T2,`T3]):T3;
  fun triple(x:`T1, y:`T2, z:`T3):triple[T1,T2,T3];  -- create a triple
extend class triple[`T1 <= comparable[T1],
                    `T2 <= comparable[T2],
                    `T3 <= comparable[T3]]
        isa comparable[triple[T1,T2,T3]];
extend class triple[`T1 <= partially_ordered[T1],
                    `T2 <= partially_ordered[T2],
                    `T3 <= partially_ordered[T3]]
        isa partially_ordered_using_<=[triple[T1,T2,T3]];
extend class triple[`T1 <= hashable[T1],
                    `T2 <= hashable[T2],
                    `T3 <= hashable[T3]]
        isa hashable[triple[T1,T2,T3]];
fun <=_lex(p1:triple[`T11 <= `T1, `T12 <= `T2, `T13 <= `T3],
           p2:triple[`T21 <= `T1, `T22 <= `T2, `T23 <= `T3]):bool
                  where `T1 <= ordered[T1],
                        `T2 <= ordered[T2],
                        `T3 <= ordered[T3];

end module Triple;
```

Quadruples are also provided.

```
module Quadruple;
class quadruple[T1,T2,T3,T4]
        isa quadruple[`S1 >= T1, `S2 >= T2, `S3 >= T3, `S4 >= T4];
  field first (:quadruple[`T1,`T2,`T3,`T4]):T1;
  field second(:quadruple[`T1,`T2,`T3,`T4]):T2;
  field third (:quadruple[`T1,`T2,`T3,`T4]):T3;
  field fourth(:quadruple[`T1,`T2,`T3,`T4]):T4;
  fun quadruple(x:`T1, y:`T2, z:`T3, w:`T4):quadruple[T1,T2,T3,T4];  -- create a quadruple
extend class quadruple[`T1 <= comparable[T1],
                       `T2 <= comparable[T2],
                       `T3 <= comparable[T3],
                       `T4 <= comparable[T4]]
        isa comparable[quadruple[T1,T2,T3,T4]];
extend class quadruple[`T1 <= partially_ordered[T1],
                       `T2 <= partially_ordered[T2],
                       `T3 <= partially_ordered[T3],
                       `T4 <= partially_ordered[T4]]
        isa partially_ordered_using_<=[quadruple[T1,T2,T3,T4]];
extend class quadruple[`T1 <= hashable[T1],
                       `T2 <= hashable[T2],
                       `T3 <= hashable[T3],
                       `T4 <= hashable[T4]]
        isa hashable[quadruple[T1,T2,T3,T4]];
fun <=_lex(p1:quadruple[`T11<=`T1, `T12<=`T2, `T13<=`T3, `T14<=`T4],
           p2:quadruple[`T21<=`T1, `T22<=`T2, `T23<=`T3, `T24<=`T4]
          ):bool where `T1 <= ordered[T1],
                       `T2 <= ordered[T2],
                       `T3 <= ordered[T3],
                       `T4 <= ordered[T4];
```

```
end module Quadruple;
```

Quintuples are also provided.

```
module Quintuple;
class quintuple[T1,T2,T3,T4,T5]
        isa quintuple[`S1 >= T1, `S2 >= T2, `S3 >= T3, `S4 >= T4, `S5 >= T5];
  field first (:quintuple[`T1,`T2,`T3,`T4,`T5]):T1;
  field second(:quintuple[`T1,`T2,`T3,`T4,`T5]):T2;
  field third (:quintuple[`T1,`T2,`T3,`T4,`T5]):T3;
  field fourth(:quintuple[`T1,`T2,`T3,`T4,`T5]):T4;
  field fifth (:quintuple[`T1,`T2,`T3,`T4,`T5]):T5;
  fun quintuple(x:`T1, y:`T2, z:`T3, w:`T4, v:`T5
                       ):quintuple[T1,T2,T3,T4,T5];  -- create a quintuple
extend class quintuple[`T1 <= comparable[T1],
                              `T2 <= comparable[T2],
                              `T3 <= comparable[T3],
                              `T4 <= comparable[T4],
                              `T5 <= comparable[T5]]
        isa comparable[quintuple[T1,T2,T3,T4,T5]];
extend class quintuple[`T1 <= partially_ordered[T1],
                              `T2 <= partially_ordered[T2],
                              `T3 <= partially_ordered[T3],
                              `T4 <= partially_ordered[T4],
                              `T5 <= partially_ordered[T5]]
        isa partially_ordered_using_<=[quintuple[T1,T2,T3,T4,T5]];
extend class quintuple[`T1 <= hashable[T1],
                              `T2 <= hashable[T2],
                              `T3 <= hashable[T3],
                              `T4 <= hashable[T4],
                              `T5 <= hashable[T5]]
        isa hashable[quintuple[T1,T2,T3,T4,T5]];
fun <=_lex(p1:quintuple[`T11<=`T1, `T12<=`T2, `T13<=`T3,
                               `T14<=`T4, `T15<=`T5],
                 p2:quintuple[`T21<=`T1, `T22<=`T2, `T23<=`T3,
                               `T24<=`T4, `T25<=`T5]
                 ):bool where `T1 <= ordered[T1],
                               `T2 <= ordered[T2],
                               `T3 <= ordered[T3],
                               `T4 <= ordered[T4],
                               `T5 <= ordered[T5];


end module Quintuple;
```

3 Control structures

3.1 Booleans and branching

In `boolean.diesel`:

The `bool` type is the type of boolean values. There are two constants of `bool` type, `true` and `false`. Boolean values are comparable and hashable, but more importantly they support a number of basic control structures.

```
extend module Stdlib;
```

```
module Bool;


predefined abstract class bool;

extend class bool isa identity_hashable[bool];


predefined object true isa bool;


predefined object false isa bool;
```

`if` takes either one closure or two closure arguments, acting like an if-then or an if-then-else statement, as follows:

```
if(i < j, {
    -- then statements
}, {
    -- else statements
});
```

The three-argument `if` control construct returns the value of the executed closure; i.e., if-then-else can be used as an expression, not just a statement, like the `? :` expression in C. The two-argument versions ignore the result of their argument closure.

`if_false` negates the result of the test.

One way to achieve the effect of an if-then-elseif-then-...-else construct is to use another `if` call as the body of the else; unfortunately, this tends to indent poorly. The `switch` construct (which really is more like Lisp `cond`) indents better but may not run as fast.

The compiler implementation nearly always implements `if` and `if_false` as efficiently as you'd expect in C, despite its source-level use of messages and closures.

```
fun if(:bool, tc:&():‘T, fc:&():‘T):T;
fun if(t:bool, tc:&():void):void;
fun if_false(t:bool, fc:&():‘T, tc:&():‘T):T;
fun if_false(t:bool, fc:&():void):void;
```

Other control structures on booleans include `&` (and), `|` (or), `not`, and `=>` (implies). These control structures' second argument is either a boolean expression or a closure with boolean return type. When the second argument is not a closure, it is always evaluated. To achieve short-circuiting semantics, use closure arguments. For example:

```
if(x != 0 & { y / x > 0 }, {
    ...
});
```

```
fun |(:bool, :bool | &():bool):bool;
fun &(:bool, :bool | &():bool):bool;
fun not(:bool):bool;
fun =>(l:bool, r:bool | &():bool):bool;  -- not(l) or r
```

The `assert` method allows checks to be made in support of defensive programming. If the test expression evaluates to is `false`, the program quits with an error message. The closure form is provided in case constructing the error message is expensive, and is short-circuited in the (expected) case where the test is true. (There currently is no mechanism to disable assertion checking, selectively or otherwise, other than editing the implementation of assert to be empty, in which case the compiler will merrily remove all calls to `assert`.)

```
fun assert(b:bool):void;   -- Standard failure message
fun assert(:bool, :string | &():string):void;  -- User-specified failure message
```

The as_integer method returns 0 for false and 1 for true.

```
fun as_integer(:bool):int;

end module Bool;
```

3.2 Looping and closures

In closure.diesel:

```
extend module Stdlib;

module Closure;
```

The predefined closure abstract class is the abstract superclass of all closure objects. Methods that dispatch on closure types (e.g. `method f(x@&(int):bool):void {...}`) actually are dispatching on this object. User-defined objects can inherit from closure in order to inherit closure-manipulating functions.

```
predefined abstract class closure;
```

Closure types follow the standard contravariant subtyping relationship among closure types, i.e.:

```
predefined type &(T1, ..., TN):T isa closure, &(‘S1 <= T1, ..., ‘SN <= TN):‘S >= T
```

The loop function invokes its closure argument endlessly. It never returns normally. To exit the loop, the closure must do a non-local return or invoke some closure that does (e.g., using the exit control structure). All other looping constructs are built upon this function.

```
fun loop(c:&():void):none;
```

while implements a standard while-do loop. E.g.:

```
while({ i < last }, {
          ...
          i := i.succ;
});
```

```
fun while(cond:&():bool, c:&():void):void;
```

Other while-do and do-until loops are implemented using the {while,until}[_{true,false}] functions. The one-argument while_{true,false} functions simply evaluate their argument test until it returns true or false, respectively, presumably for its side effects. The until_ versions evaluate their body closure and then the test until the test returns true or false, as appropriate.

```
fun while_true(cond:&():bool, c:&():void):void;  -- same as while
fun while_false(cond:&():bool, c:&():void):void;
fun while(cond:&():bool):void;
fun while_true(cond:&():bool):void;  -- same as while
fun while_false(cond:&():bool):void;
fun until(c:&():void, cond:&():bool):void;
fun until_true(c:&():void, cond:&():bool):void;  -- same as until
fun until_false(c:&():void, cond:&():bool):void;
```

The exit[_value][_continue] constructs support evaluating a block of code (the body closure), breaking out of it if the break closure is evaluated inside body. The _value versions return a value. The _continue versions allows body to be restarted from the beginning when the continue closure is evaluated.

For example, to execute some code, but perhaps quit early:

```
exit(&(break:&():none){
    ...
    if(..., { eval(break); }); -- skip the rest of the body of exit
    ...
    -- fall off bottom
});
```

This idiom supports breaking out of any sort of looping or non-looping piece of code: just wrap the thing in an exit or exit_value control structure and invoke the break block where the control structure should be exited.

```
fun exit(c:&(exit:&():none):void):void;
fun exit_value(c:&(exit:&(‘T):none):‘T):T;
fun exit_continue(c:&(exit:&():none, continue:&():none):void):void;
fun exit_value_continue(c:&(exit:&(‘T):none, continue:&():none):T):T;
```

The loop[_exit[_value]][_continue] functions evaluate the body closure again and again until the break closure is evaluated inside body. For the _continue versions, execution of body can be restarted from the beginning by evaluating the continue closure. The _value versions return a value.

To write a simple loop with a break statement:

```
loop_exit(&(break:&():none){
    ...
    if(..., { eval(break); }); -- exit loop conditionally
    ...
    -- loop
});
```

To loop and compute a value:

```
let result:int := loop_exit_value(&(break:&(int):none){
    ...
    if(..., { eval(break, theResult) }); -- exit loop, returning theResult
    ...
});
```

If both break and continue are desired for an arbitrary iterating construct, such as times_do, two exit calls should be used, as in the following example. The outer call encloses the iterator and provides breaking out of the loop, while the inner call encloses the loop body and provides continuing to the next iteration:

```
exit(&(break:&():none){
    10.times_do(&(i:int){                -- for i := 0 to 10-1 do
        exit(&(continue:&():none){
            ...
            if(..., break);          -- break out of loop
            ...
            if(..., continue);       -- continue the iteration by jumping
                                     -- to the end of the loop body
            ...
        });
    });
});
```

Some standard iteration control structures, including `do` and `do_associations`, support break and continue functionality more conveniently via `do[_associations][_exit][_continue]` alternatives. For example, the following code iterates through the `coll` collection, also allowing the body closure to exit the iteration or continue the iteration with the next element:

```
coll.do_exit_continue(&(elem:T, break:&():none, continue:&():none){
    ...
    if(..., break);         -- break out of loop
    ...
    if(..., continue);      -- continue the iteration with the next element
    ...
});
```

```
fun loop_exit(c:&(exit:&():none):void):void;
fun loop_exit_value(c:&(exit:&(‘T):none):void):T;
fun loop_exit_continue(c:&(exit:&():none, continue:&():none):void
                             ):void;
fun loop_exit_value_continue(c:&(exit:&(‘T):none,
                                       continue:&():none):void
                                   ):T;
fun loop_continue(c:&(continue:&():none):void):void;
```

Case statements are supported through the `switch`, `case`, and `else` functions. The elements of `switch`'s argument collection (usually a vector literal expression) are evaluated in turn, until one of the test blocks evaluates to `true` or the `else` case is found. Then the corresponding `do` block is evaluated and its result returned as the result of the `switch` function. (Thus `switch` is very much like Lisp's `cond`.) The `switch` function dies with a run-time error if none of the cases matches and there is no `else` case. To illustrate:

```
let result:string :=
  switch([case({ x < 0 }, { "negative" }),
          case({ x = 0 }, { "zero" }),
          else(           { "positive" })]);
```

Unfortunately, unlike most Diesel control structures, the `switch` construct is not as efficient as the corresponding C version: a vector object is created and filled in with objects containing real closures, and a bunch of messages get sent. So you might wish to use chained `if` expressions instead of `switch` expressions in the most time-critical parts of your program. (Some optimizations have been implemented that often transform `switch` statements of this form into an if-then-else chain, but only if you invoke `unrolled_switch` instead of `switch`, and object creations still remain unless `debug_support` is disabled.)

```
class case_pair[T] isa case_pair[‘S >= T];
fun case(c:&():bool, s:&():‘T):case_pair[T];
fun else(s:&():‘T):case_pair[T];
fun switch(t:ordered_collection[case_pair[‘T]]):T;
fun unrolled_switch(t:i_vector[case_pair[‘T]]):T;   -- an optimized version for short vector literal args

end module Closure;
```

3.3 Exception handling

In `error.diesel`:

```
extend module Stdlib;

module Error;
```

The `handle_system_errors` function executes its argument closure. Any Diesel "system errors" (e.g., `error` or `exit` calls) caused during evaluation of the closure are caught and suppressed (after the trap into the debugger returns). The boolean return value indicates whether an error was suppressed.

```
fun handle_system_errors(cl:&():void):bool;
```

The `unwind_protect` function allows a "clean up" closure to be executed whenever control returns through the `unwind_protect` function call, either normally, via a non-local return, or via a Diesel "system error." `unwind_protect` first invokes its `cl` closure argument. When control returns from this invocation, the `on_return` closure is invoked. If the `on_return` closure returns normally, the returning of the `cl` closure is resumed: if the `cl` closure returned normally, the result of this closure is returned normally as the result of the `unwind_protect` function; otherwise it continues to throw whatever exception or system error the body did. If the `on_return` closure returns abnormally, either via a non-local return or a system error, then this result supercedes the original suspended result of the `cl` closure. (`unwind_protect` is similar to the like-named construct in Common Lisp.)

```
fun unwind_protect(cl:&():‘T, on_return:&():void):T;
```

The `on_error` function allows a "clean up" closure to be executed whenever control returns through the `on_error` function call abnormally, either via a non-local return or via a Diesel "system error." `on_error` first invokes its `cl` closure argument. If control returns normally from `cl`, then `on_error` returns normally the result of this closure, without invoking its `on_error` closure. If, on the other hand, control returns from `cl` either via a non-local return or a system error, the `on_error` closure is invoked. If this closure completes normally, the abnormal returning of the `cl` closure is resumed (either continuing the non-local return or the system error raised by `cl`). If the `on_error` closure returns abnormally, either via a non-local return or a system error, then this result supercedes the original suspended abnormal result of the `cl` closure. `on_error` is like `unwind_protect`, except that the `on_error` block is only run if the `cl` block has an abnormal result.

```
fun on_error(cl:&():‘T, on_error:&():void):T;

end module Error;
```

4 Collections

In `collection.diesel`:

Collections are groups of elements. We first introduce abstract interfaces, describing and refining various kinds of collections and their operations. We then shift to discussing concrete implementations which can be instantiated and manipulated at runtime.

There are three major families of collections: unordered collections (like bags and sets), ordered collections (e.g., lists), and tables (or keyed collections). Indexed collections like arrays, vectors, and strings are both ordered collections and keyed tables, where the keys are integer indices.

Abstract collection classes typically come in three versions: read-only, immutable, and mutable, with the latter two indicated by `i_` and `m_` prefixes. Moreover, the read-only and immutable versions come in "generic" and "exactly" subversions. For example:

```
abstract class vector[‘T] isa indexed[T], vector[‘S >= T];
abstract class vector_exactly[‘T] isa indexed_exactly[T], vector[‘T];
abstract class i_vector[‘T] isa i_indexed[T], i_vector[‘S >= T];
abstract class i_vector_exactly[‘T] isa i_indexed_exactly[T], i_vector[‘T];
abstract class m_vector[‘T] isa m_indexed[T], vector_exactly[T];
```

These five versions of each abstraction is a standard idiom among the collection classes. A read-only base class (e.g., `vector`) defines the read-only behavior of this kind of collection, but doesn't specify whether or not a particular value of this type is mutable. One subclass (e.g., `i_vector`) adds the immutability specification; while the immutable variety might not add any new operations compared to the read-only interface, it does guarantee that the collection does not change after it is created. Another subclass (e.g., `m_vector`) adds the mutator operations. (An immutable collection may contain mutable objects which may be side-effected while in the collection, but the collection itself cannot be changed.) Each of these

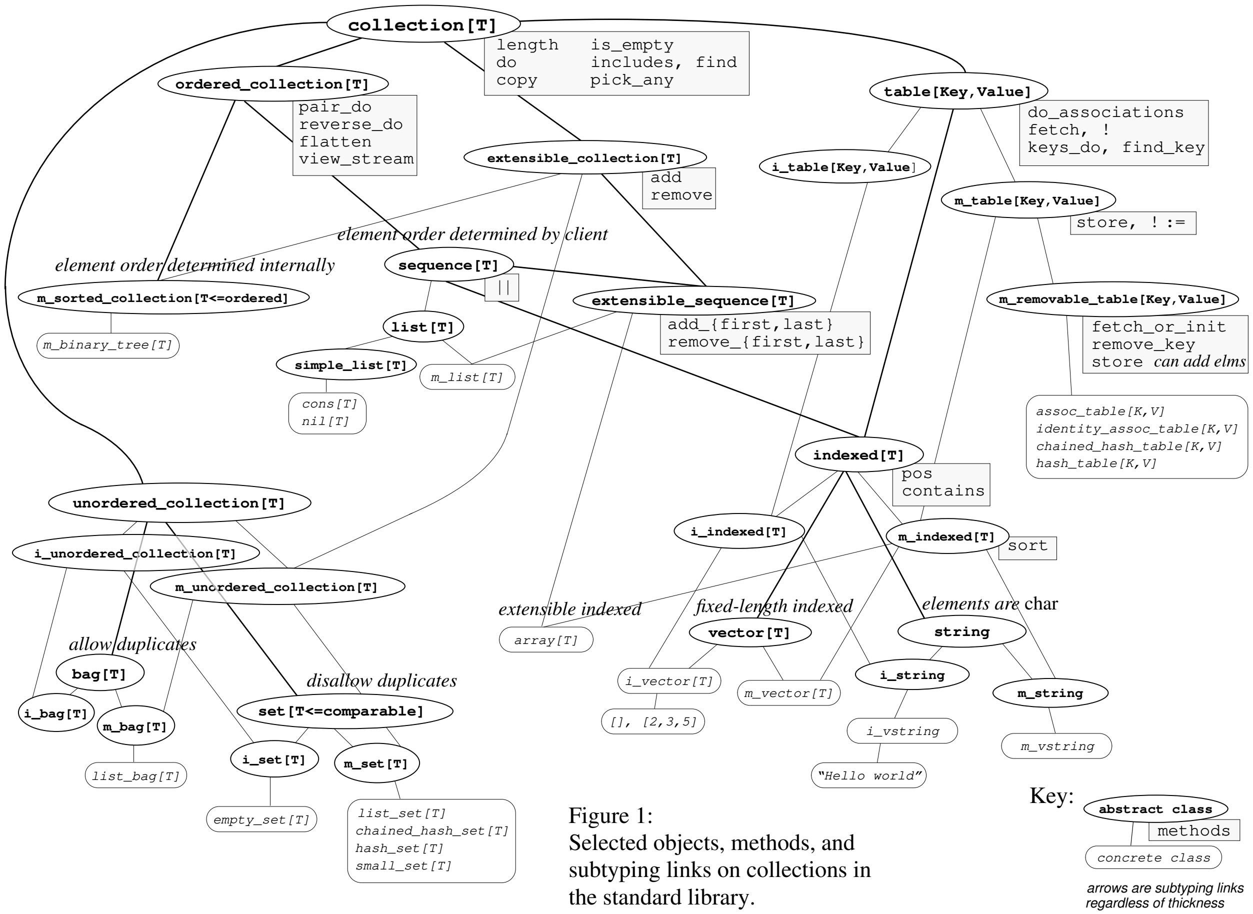

Figure 1:
Selected objects, methods, and subtyping links on collections in the standard library.

versions inherits from the corresponding version(s) of its superclass(es), e.g. `i_vector[T]` inherits from `i_indexed[T]` while `m_vector[T]` inherits from `m_indexed[T]`. (Some abstractions omit one or both of the immutable or mutable subclasses.)

Separating the read-only from the read-write interfaces (e.g., `vector` vs. `m_vector`) allows the read-only versions to support richer subtyping relationships. For example, a `vector[T]` object is a read-only vector of objects, all of which are subtypes of `T`. Such a collection can be safely viewed as a `vector[S]` for any `S` that is a supertype of `T`, since such a collection is also a read-only vector of objects, all of which are subtypes of `S`. For example, `vector[int]` is a subtype of `vector[num]`. We say that the `vector` type is covariant in its parameter type, and this covariance is indicated by the `vector['T] isa ..., vector['S >= T]` declaration.

Whether or not a class can safely be declared to be covariant (or contravariant) in a type parameter is determined by how functions that take an argument of that class take other arguments that also mention the type parameter. Mentions can be "positive" or "negative", where the result type is positive, an argument type is negative, and polarity reverses inside an argument of a closure type. Covariant type parameters can only appear in positive positions (like results and arguments of argument closures), while contravariant type parameters can only appear in negative positions (like arguments). Type parameters that appear in both positive and negative positions across all the functions taking values of collections of that type parameter can be neither covariant nor contravariant. (The truth is somewhat more subtle than this, but it's true to a first approximation.)

Read-only collections like `vector[T]` only support functions all of whose occurrences of `T` appear in positive positions, and so their type paraters are covariant. This includes most (but not all) reader operations, like `fetch` and `do`. Mutable collections support functions whose occurrences occur in both positive (e.g. `fetch`) and negative (e.g. `store`) positions, so their type parameters are not covariant. For example, `m_vector[int]` is unrelated to `m_vector[num]`. Indeed, `m_vector[int]` cannot have floats stored in it (unlike `m_vector[num]`), and so `m_vector[int]` is not a subtype of `m_vector[num]`. Also, `m_vector[num]` can contain things other than `int`s and reveal this through `do`, `pick_any`, etc. (unlike `m_vector[int]`), so the reverse subtyping relation doesn't hold, either.

A few reader operations have type parameters in negative positions, such as `includes` and `=`. Consequently (to a first approximation), such operations cannot be supported by the generic read-only collections like `vector[T]` with covariant type parameters. To allow them, the `_exactly` versions of read-only collections are introduced. The type parameter of an `_exactly` collection is not covariant, and so it can support arbitrary operations. E.g. `vector_exactly[T]` is a subtype of `vector[T]` (and therefore also `vector[S]` for any `S` that's a supertype of `T`) but not `vector_exactly[S]` for any `S` other than `T`. (There's no need for a `m_vector_exactly[T]` type, since `m_vector[T]` already is not covariant.)

To support the most reuse, then, the read-only non-exactly interface should be used as a type declaration whenever possible. Only if mutation, immutability, or certain comparison operations are required should the mutable, immutable, or `_exactly` subtype interface be used. Observe that the immutable interface must be distinguished from the read-only interface. Indeed, a mutable collection subtypes the read-only interface, but not the immutable interface (which would violate the behavioral guarantee of immutability).

There are three common varieties of mutation: replacement (removing one element but inserting another in its place), insertion (increasing the collection's size), and deletion (making the collection smaller). The latter two capabilities are captured by the abstract `extensible_collection[T]` and `removable_collection[T]` classes, which may be inherited by mutable collections.

4.1 Basic collections

```
extend module Stdlib;
```

```
module Collection;
```

`collection[T]` is a collection of items of some type `T` (or any subtype of `T`). A collection of some type `T` is a subtype of all collections of types that are supertypes of `T`.

```
abstract class collection[T] isa collection[‘S >= T];
abstract class collection_exactly[T] isa collection[T];
```

Collections support a `length` operation (implemented by subclasses) which returns the number of elements in the collection, plus a number of length-related predicates.

```
fun length(:collection[‘T]):int;
fun is_empty(c:collection[‘T]):bool;  -- length = 0
fun non_empty(c:collection[‘T]):bool;  -- length ¿ 0
fun is_singleton(c:collection[‘T]):bool;  -- length = 1
fun is_multiple(c:collection[‘T]):bool;  -- length ¿ 0
```

Collections support a number of control structures. Primary among all control structures is the `do` method that iterates through the collection and invokes an argument closure on each element. Elements are processed in some unspecified order which may vary from invocation to invocation, even if the collection is not modified between finishing one `do` loop and starting the next. For example:

```
  myCollection.do(&(elem:elemType){      -- bind elem to each element
      ...                                -- of myCollection in turn
  });
```

```
fun do(:collection[‘T], closure:&(T):void):void;
```

The collection cannot be modified while `do` is active without potentially bizarre results. A related control structure, `do_allowing_updates`, allows the collection to be modified during the iteration. (The effect of modification during iteration depends on the kind of collection and the kind of update.)

```
fun do_allowing_updates(t:collection[‘T], closure:&(T):void):void;
```

The `do_exit` iterator passes an additional argument to its argument closure, which can be invoked to exit the `do` loop prematurely, analogously to the `loop_exit` control structure. The `do[_exit]_continue` iterators are similarly analogous.

```
fun do_exit(t:collection[‘T], closure:&(T, &():none):void):void;
fun do_continue(t:collection[‘T], closure:&(T, &():none):void):void;
fun do_exit_continue(t:collection[‘T],
                            closure:&(T, &():none, &():none):void):void;
```

Two collections of comparable elements can be compared to see if they have the same elements, ignoring order, using `=_unordered`.

```
fun =_unordered(c1:collection_exactly[‘T],
                      c2:collection_exactly[‘T]):bool
                                        where T <= comparable[T];
fun !=_unordered(c1:collection_exactly[‘T],
                       c2:collection_exactly[‘T]):bool
                                        where T <= comparable[T];
```

The `includes` method computes whether the collection `c` contains an element which is equal, using `=`, to `x`. The `includes_all` method returns `true` if `c` contains elements which are equal to all the elements of `c2`. The `includes_some` method returns `true` if `c` contains an element which satisfies the predicate `test`.

```
fun includes(c:collection_exactly[‘T], x:T):bool where =(:T,:T):bool;
fun includes_all(c1:collection_exactly[‘T],
                       c2:collection_exactly[‘T]):bool where =(:T,:T):bool;
fun includes_generically(c:collection[‘T1 <= ‘T], x:T):bool
                                                     where =(:T,:T):bool;
fun includes_all_generically(c1:collection[‘T1 <= ‘T],
                                   c2:collection[‘T2 <= ‘T]):bool
                                                     where =(:T,:T):bool;
fun includes_some(c:collection[‘T], test:&(T):bool):bool;
```

The `count` method returns the number of times an element appears in a collection.

```
fun count(c:collection_exactly[‘T], x:T):int where T <= comparable[T];
fun count_generically(c:collection[‘T1 <= ‘T], x:T):int
                                               where T <= comparable[T];
```

The `count_pred` method returns the number of elements of the collection `c` for which the predicate `test` evaluates to true.

```
fun count_pred(c:collection[‘T], test:&(T):bool):int;
```

The `find` method returns an element of collection `c` satisfying the predicate `test`.

```
fun find(c:collection[‘T], test:&(T):bool):T;
fun find(c:collection[‘T], test:&(T):bool, if_absent:&():‘S):T|S;
```

Tests whether the predicate is true of `every` or `any` collection element. (`any` is the same as `includes_some`.)

```
fun every(c:collection[‘T], test:&(T):bool):bool;
fun any(c:collection[‘T], test:&(T):bool):bool;
```

Implements the classic functional reduce operation over collections. Order of reduction is in the collection's order of iteration (i.e., left-to-right for ordered collections, unspecified for unordered collections).

```
fun reduce(t:collection[‘T], bin_op:&(T,S):S, init:‘S):S;
```

Implements a streamlined version that works only on nonempty collections, invoking a closure on an empty collection, and doesn't require an init value.

```
fun reduce(t:collection[‘T], bin_op:&(T,‘S >= T):S):S;
fun reduce_nonempty(t:collection[‘T], bin_op:&(T,‘S >= T):S):S;
fun reduce_nonempty(t:collection[‘T], bin_op:&(T,‘S >= T):S,
                           if_empty:&():‘E):S|E;
```

The `min`, `max`, and `average` methods are defined on collections as well as pairs of values. They are synonyms of the `(min|max|average)_over_all` methods, which optionally take a closure to handle empty conditions.

```
fun min(t:collection[‘T <= ordered[T]]):T;
fun min_over_all(t:collection[‘T <= ordered[T]]):T;
fun min_over_all(t:collection[‘T <= ordered[T]], if_empty:&():‘S):T|S;
fun max(t:collection[‘T <= ordered[T]]):T;
fun max_over_all(t:collection[‘T <= ordered[T]]):T;
fun max_over_all(t:collection[‘T <= ordered[T]], if_empty:&():‘S):T|S;
fun average(t:collection[‘T <= num]):T;
fun average_over_all(t:collection[‘T <= num]):T;
fun average_over_all(t:collection[‘T <= num], if_empty:&():‘S):T|S;
fun total(t:collection[‘T <= num]):T|int;
```

The `pick_any` method returns some element of the collection, invoking `if_empty` or producing an error if the collection is empty. The `only` method returns the only element of the argument collection, producing an error or invoking `if_non_singleton` if the collection has zero or multiple elements.

```
fun pick_any(c:collection[`T]):T;
fun pick_any(c:collection[`T1], if_empty:&():`T2):T1|T2;
fun only(c:collection[`T]):T;
fun only(c:collection[`T1], if_non_singleton:&():`T2):T1|T2;
fun select(c:collection[`T], pred:&(T):bool):collection[T];
fun select_first(c:collection[`T], howmany:int):collection[T];
fun select_first(c:collection[`T], hmy:int,
                        pred:&(T):bool):collection[T];
fun select_as_m_list(c:collection[`T], howmany:int, pred:&(T):bool
                            ):m_list[T];
fun select_as_array(c:collection[`T], howmany:int, pred:&(T):bool
                            ):array[T];
fun select_as(c:collection[`T], howmany:int, pred:&(T):bool,
                     result:`R <= extensible_sequence[T]):R;
```

Collections can be copied. This copy is a shallow copy; the elements of the collection are not copied. If the collection is immutable, the copy function usually returns the collection itself without doing a copy.

```
fun copy(:collection[`T]):collection[T];
```

Various files define operations like `as_vector`, `as_m_indexed`, etc., for "downcasting" and/or converting a collection to a specific kind. Note that if the collection is already of that kind, no copying is done, so don't assume that `as_X` does a copy!

Various variations on printing a collection are available. The standard `print_string` behavior includes `open_brace`, `elems_print_string`, and then `close_brace`. By default `open_brace` contains the name of the collection and an open brace, `elems_print_string` consists of the elements of the collection, separated by the `elem_separator` (by default, a comma), and `close_brace` is a close brace.

```
fun collection_name(:collection[`T]):string;
fun open_brace(c:collection[`T]):string;
fun close_brace(:collection[`T]):string;
fun elems_print_string(c:collection[`T]):string;
fun elems_print(c:collection[`T]):void;
fun elem_separator(:collection[`T]):string;
fun elem_print_string(t:collection[`T], elem:T, first:bool):string;
fun elem_print(t:collection[`T], elem:T, first:bool):void;

end module Collection;
```

4.2 Removing and adding elements

In `removable.diesel`:

```
extend module Stdlib;
```

A `removable_collection` supports in-place removal of elements. The `remove` function removes one element equal to its second argument, or raises an error if not found (also defined is a variation that takes a closure to handle the not-found case). The `remove_if` function removes all elements that satisfy the test, and returns the number of elements removed. The `remove_any` function removes and returns a single arbitrary element of the collection or raises an error if the collection is empty (again, a variation on `remove_any` passes a closure to handle this case in a client-defined manner). The `remove_all` function removes all elements of the collection. Subclasses need to implement these functions; a default `remove_all` is provided, but it is likely that subclasses can provide a much more efficient implementation.

```
module RemovableCollection;
abstract class removable_collection[T] isa collection_exactly[T];
fun remove(:removable_collection[`T], x:T, if_absent:&():void):void;
fun remove(c:removable_collection[`T], x:T):void;
fun remove_if(c:removable_collection[`T], pred:&(T):bool):int;
fun remove_any(c:removable_collection[`T], if_empty:&():`S):T|S;
fun remove_any(c:removable_collection[`T]):T;
fun remove_all(c:removable_collection[`T]):void;

end module RemovableCollection;
```

In `extensible.diesel`:

```
extend module Stdlib;
```

The `functionally_extensible_collection` class generalizes collections that are extensible in place (i.e., `add` mutates) and those that are extensible, but not in place (i.e., `add` returns a new collection). The `add_functional` operation adds an element to the collection (in some unspecified location), returning a collection with the element added. This returned collection may either be the receiver collection (if the `add` is performed in-place) or some new collection. Since the receiver collection may or may not be changed and may or may not contain the new element, the caller should use the returned value (and should not use the argument to `add_functional` after the call).

The weird name of the collection is intended to suggest a functional language where `add` returns a new collection.

```
module FunctionallyExtensibleCollection;
abstract class functionally_extensible_collection[T]
                                                isa collection_exactly[T];
  fun add_functional(:functionally_extensible_collection[`T], x:T
                            ):functionally_extensible_collection[T];

end module FunctionallyExtensibleCollection;

module FunctionallyExtensibleRemovableCollection;
```

A blend of `functionally_extensible_collection` and `removable_collection`.

```
abstract class functionally_extensible_removable_collection[T]
        isa removable_collection[T], functionally_extensible_collection[T];

end module FunctionallyExtensibleRemovableCollection;
```

An `extensible_collection` supports adding new elements in-place, as well as removing elements. The `add` function adds a new element to the collection (in some unspecified place). The `add_nonmember` function can be used if the element being added is known not to be in the collection already. Its effect is that same as that of `add` for collections which allow duplicates, but it may be faster than a generic `add` for collections that do not permit duplicates (such as `set` and its subclasses). The `add_all` and `add_all_nonmember` functions support adding all the elements of some other collection to the receiver collection.

```
module ExtensibleCollection;
abstract class extensible_collection[T]
        isa functionally_extensible_removable_collection[T];
fun add(:extensible_collection[`T], x:T):void;
fun add_nonmember(c:extensible_collection[`T], x:T):void;
fun add_all(c:extensible_collection[`T], xs:collection[T]):void;
fun add_all_nonmember(c:extensible_collection[`T],
                             xs:collection[T]):void;
```

`copy_empty` produces a new, empty extensible collection of a similar kind to the receiver collection

```
fun copy_empty(:extensible_collection['T]):extensible_collection[T];

end module ExtensibleCollection;
```

4.3 Unordered collections

In `unordered.diesel`:

An `unordered_collection` is a group of elements in no particular order. It supports collection operations such as `length`, `is_empty`, `do`, `pick_any`, `includes`, `find`, and `copy`.

```
extend module Stdlib;

module UnorderedCollection;
abstract class unordered_collection[T]
               isa collection[T], unordered_collection['S >= T];
abstract class unordered_collection_exactly[T]
               isa collection_exactly[T], unordered_collection[T];
```

If the elements of the collection are comparable, then so is the collection. Two such collections are equal (`=`) if they have the same elements (with the same number of occurrences), independent of order. (By contrast, two ordered collections are equal only if they contain the same elements in the same order.) If the elements of the collection are hashable, then so is the collection.

```
  extend class unordered_collection_exactly['T <= comparable[T]]
               isa comparable[unordered_collection_exactly[T]];
  extend class unordered_collection_exactly['T <= hashable[T]]
               isa hashable[unordered_collection_exactly[T]];
```

Unordered collections support several functional set-operations which, given argument collections, return a new collection.

`union`: form the collection whose element counts are the maximum of the element counts of the argument collections

```
fun union(m1:unordered_collection_exactly['T <= comparable[T]],
                m2:unordered_collection_exactly[T]
                ):m_unordered_collection[T];
```

`intersection`: form the collection whose element counts are the minumum of the element counts of the argument collections

```
fun intersection(m1:unordered_collection_exactly['T <= comparable[T]],
                       m2:unordered_collection_exactly[T]
                       ):m_unordered_collection[T];
```

`difference`: form the collection whose element counts are the element counts of the first collection minus the element counts of the second collection (with a minimum count of 0)

```
fun difference(m1:unordered_collection_exactly['T <= comparable[T]],
                     m2:unordered_collection_exactly[T]
                     ):m_unordered_collection[T];
```

`is_disjoint`: are there no elements in common?

```
fun is_disjoint(m1:unordered_collection_exactly['T <= comparable[T]],
                      m2:unordered_collection_exactly[T]):bool;
```

`overlaps`: are there any elements in common?

```
fun overlaps(m1:unordered_collection_exactly[‘T <= comparable[T]],
                    m2:unordered_collection_exactly[T]):bool;
```

`is_subset`: are all the elements of m1 also in m2?

```
fun is_subset(m1:unordered_collection_exactly[‘T <= comparable[T]],
                     m2:unordered_collection_exactly[T]):bool;
```

`is_strict_subset`: is a strictly smaller subset?

```
fun is_strict_subset(m1:unordered_collection_exactly[‘T <= comparable[T]],
                            m2:unordered_collection_exactly[T]):bool;
```

Two refinements of `unordered_collection` indicate whether the collection is known to be immutable (`i_unordered_collection`) or mutable (`m_unordered_collection`).

An immutable unordered collection of some type `T` is a subtype of any immutable unordered collection of a supertype of `T`. In contrast, a mutable unordered collection of a type `T` has no subtyping relation to a mutable unordered collection of a different type. Both kinds are subtypes or a generic unordered collection of `T` or transitively any supertype of `T`.

```
abstract class i_unordered_collection[T]
        isa unordered_collection[T], i_unordered_collection[‘S >= T];
abstract class i_unordered_collection_exactly[T]
        isa unordered_collection_exactly[T], i_unordered_collection[T];
```

Mutable collections provide remove, add, and other operations of `extensible_collection`. Method `copy_empty` always returns an empty mutable collection with a type like that of its argument.

```
abstract class m_unordered_collection[T]
        isa unordered_collection_exactly[T], extensible_collection[T];
```

```
end module UnorderedCollection;
```

4.3.1 Sets

In `set.diesel`:

```
extend module Stdlib;
```

```
module Set;
```

Sets are a specialization of unordered collections that explicitly disallow duplicates.

```
abstract class set[T <= comparable[T]]
                                        isa unordered_collection_exactly[T];
```

Sets refine the unordered collection set-like operations to return sets, not just plain unordered collections.

```
fun copy_mutable(:set[‘T]):m_set[T];
abstract class i_set[T <= comparable[T]]
                                isa set[T], i_unordered_collection_exactly[T];
```

One standard implementation of an immutable set is the concrete object `empty_set`.

```
object empty_set[T <= comparable[T]] isa i_set[T];
```

Mutable sets can be added to.

```
abstract class m_set[T <= comparable[T]] isa set[T],
                                                    m_unordered_collection[T],
                                                    removable_collection[T];
  fun check_if_missing_and_add(m:m_set[‘T], x:T):bool;
```

4.3.2 Set implementations

```
let var warn_for_list_sets_longer_than:int;
```

list_set is an implementation of m_set using a linked list as the core representation.

```
class list_set[T <= comparable[T]] isa m_set[T];
  fun new_list_set[T <= comparable[T]]():list_set[T];
  fun as_list_set(c:collection_exactly[‘T <= comparable[T]]):m_set[T];

end module Set;
```

In hash-set.diesel:

An m_set implementation using an open hashing algorithm. Set elements must be hashable.

The new_[chained_]hash_set methods optionally take a max_size argument, which is the expected maximum size of the set; the initial size of all newly-created sets is 0. All hashing set implementations automatically resize if the set grows too large or small. Warning: the do_allowing_updates function on hash_set (and hash_table and hash_keyed_set, the other open-hash-table-based implementations) may not support more than one add (or store or fetch_or_init or any operation that increases the size of the collection) during the iteration, because they are blocked from resizing but they may need to resize to support multiple adds.

```
extend module Stdlib;

module HashSet;
class hash_set[T <= hashable[T]] isa m_set[T];
fun new_hash_set[T <= hashable[T]]():hash_set[T];
fun new_hash_set[T <= hashable[T]](size:int):hash_set[T];
fun copy_as_hash_set(src:collection[‘T <= hashable[T]]):hash_set[T];
fun copy_as_hash_set[T](src:collection[‘T <= hashable[T]]):hash_set[T];
fun copy_as_hash_set[T](src   :collection[‘T <= hashable[T]],
                               length:int
                               )     :hash_set[T];
fun new_hash_set_from(src:collection[‘Src],
                             cl :&(Src):‘Res <= hashable[Res]
                             )  :hash_set[Res];
fun new_hash_set_from[Res <= hashable[Res]](src:collection[‘Src],
                                                 cl :&(Src):Res
                                                 )  :hash_set[Res];
fun new_hash_set_from[Res <= hashable[Res]](src   :collection[‘Src],
                                                 length:int,
                                                 cl    :&(Src):Res
                                                 )     :hash_set[Res];

end module HashSet;
```

In chained-hash-set.diesel:

```
extend module Stdlib;

module ChainedHashSet;
```

chained_hash_set is an m_set implementation using a closed hashing algorithm. Set elements must be hashable.

```
class chained_hash_set[T <= hashable[T]] isa m_set[T];
fun new_chained_hash_set[T <= hashable[T]]():chained_hash_set[T];
fun new_chained_hash_set[T <= hashable[T]](size:int):chained_hash_set[T];
```

```
end module ChainedHashSet;
```

In `small-set.diesel`:

An implementation of `m_set` that is space efficient when there are only a few elements, but scales well when there are a large number of elements. Elements of small sets are required to be `hashable`.

```
extend module Stdlib;

module SmallSet;
class small_set[T <= hashable[T]] isa m_set[T];
fun shrink_set(t:small_set[`T]):void;
fun new_small_set[T <= hashable[T]]():small_set[T];

end module SmallSet;

module AbsentSmallElement;
object absent_small_element;
  fun elem_present(:any):bool;

end module AbsentSmallElement;
```

In `bit-set.diesel`:

A `bit_set` is an abstract class for bit-vector-based set representations. Concrete subclasses of `bit_set` must provide the `element_to_index` and `index_to_element` functions mapping between elements and bit positions. If the elements store their own set indices as an `id_num` field, the interface provided by the `caching_bit_set` subclass can be used. If a separate table mapping elements to indices is needed, the `hashing_bit_set` subclass can be used.

Concrete subclasses must also provide implementations of new_bit_set to call the correct constructor.

```
extend module Stdlib;

module BitSet;
abstract class bit_set[T <= comparable[T]] isa m_set[T],
                                                    hashable[bit_set[T]];
  fun element_to_index(:bit_set[`T], :T):int;
  fun index_to_element(:bit_set[`T], :int):T;
  fun new_bit_set(:bit_set[`T], :bit_vector, cached_length:int
                        ):bit_set[T];
  fun union_in_place(a:`S <= bit_set[`T], b:`S <= bit_set[`T]):void;
  fun intersection_in_place(a:`S <= bit_set[`T],
                                 b:`S <= bit_set[`T]):void;
  fun difference_in_place(a:`S <= bit_set[`T],
                               b:`S <= bit_set[`T]):void;
  fun includes_id(a:bit_set[`T], idx:int):bool;
  fun add_id(a:bit_set[`T], idx:int):void;
class bit_set_id_manager[T];
  fun allocate_index_for_element(t:bit_set_id_manager[`T], elem:T):int;
  fun lookup_element_for_index(t:bit_set_id_manager[`T], idx:int):T;
  fun remove_element(t:bit_set_id_manager[`T], elem:T, idx:int):void;
  fun reset_id_manager(t:bit_set_id_manager[`T]):void;
  fun new_bit_set_id_manager[T]():bit_set_id_manager[T];
abstract class hashing_bit_set[T <= hashable[T]] isa bit_set[T];
class hashing_bit_set_id_manager[T <= hashable[T]]
                                                isa bit_set_id_manager[T];
  fun lookup_index_for_element(t:hashing_bit_set_id_manager[`T],
                                    elem:T):int;
  fun new_hashing_bit_set_id_manager[T <= hashable[T]]()
```

```
     :hashing_bit_set_id_manager[T];
abstract class caching_bit_set[T <= caching_bit_set_element[T]]
                                isa bit_set[T];
  fun id_manager(:caching_bit_set[`T]):bit_set_id_manager[T];
abstract class caching_bit_set_element[T <= caching_bit_set_element[T]]
                                isa hashable[caching_bit_set_element[T]];
  fun elem_id_manager(:`T <= caching_bit_set_element[T]
                             ):bit_set_id_manager[T];
  var field id_num(t:`T <= caching_bit_set_element[T]):int;
  fun reset_id_num(t:`T <= caching_bit_set_element[T]):void;
abstract class caching_bit_set_2[T <= caching_bit_set_element_2[T]]
                                                                isa bit_set[T];
abstract class caching_bit_set_element_2[
                                        T <= caching_bit_set_element_2[T]]
                                isa hashable[caching_bit_set_element_2[T]];
  fun elem_id_manager_2(:`T <= caching_bit_set_element_2[T]
                               ):bit_set_id_manager[T];
  var field id_num_2(t:`T <= caching_bit_set_element_2[T]):int;
  fun reset_id_num_2(t:`T <= caching_bit_set_element_2[T]):void;

end module BitSet;
```

4.3.3 Bags

In `bag.diesel`:

Bags are a specialization of unordered collections that explicitly allow duplicates.

```
extend module Stdlib;

module Bag;
abstract class bag[T] isa unordered_collection[T],
                                 -- type parameter is covariant
                                 bag[`S >= T];
abstract class bag_exactly[T] isa bag[T];
abstract class i_bag[T] isa bag[T], i_unordered_collection[T],
                                   -- type parameter is covariant
                                   i_bag[`S >= T];
abstract class i_bag_exactly[T]
        isa bag_exactly[T], i_unordered_collection_exactly[T], i_bag[T];
abstract class m_bag[T] isa bag_exactly[T], m_unordered_collection[T];
extend class m_bag[`T <= comparable[T]] isa removable_collection[T];
```

The `list_bag` class is a concrete implementation of the mutable bag abstraction, using a linked list as the core representation. The `new_list_bag` method is the "constructor" for this data structure.

```
class list_bag[T] isa m_bag[T];
  fun new_list_bag[T]():list_bag[T];

end module Bag;
```

In `hash-bag.diesel`:

The `hash_bag` class is a concrete implementation of the mutable bag abstraction, using a hash table mapping bag elements to an occurrence count. `new_hash_bag` method is the "constructor" for this data structure.

```
extend module Stdlib;
```

```
module HashBag;
class hash_bag[T <= hashable[T]] isa m_bag[T];
  fun set_length(:hash_bag[‘T], :int):void;
```

Hash-bags support iterating through runs of elements as a group, using `do_with_counts`. Each distinct element value is visited only once.

```
  fun do_with_counts(m:hash_bag[‘T], c:&(T,int):void):void;
  fun add_count(m:hash_bag[‘T], x:T, count:int):void;
  fun add_nonmember_count(m:hash_bag[‘T], x:T, count:int):void;
  fun new_hash_bag[T <= hashable[T]]():hash_bag[T];

end module HashBag;
```

4.3.4 Union-find sets

In `union-find-set.diesel`:

`Union_find_set` is a framework for fast union-find data structures, where two union_find_set instances can be merged into a single equivalence class (`union_set`) and the canonical equivalence class representative can be determined from any member (`find_set`), in near-linear time. The parameter refers to the specific subclass of union_find_set being manipulated, so that e.g. `find_set` on a specific subclass returns the same kind of specific subclass rather than a generic `union_find_set`.

```
extend module Stdlib;

module UnionFindSet;
abstract class union_find_set[T <= union_find_set[T]]
                                          isa comparable[union_find_set[T]];
```

find the equivalence class representative

```
fun find_set(x:‘T <= union_find_set[T]):T;
```

unify two sets

```
fun union_set(x:‘T <= union_find_set[T], y:‘T <= union_find_set[T]):T;
```

when two union_find_sets are merged, the helper function `merge_set` is called with the chosen representative as the first argument and the other representative as the other argument. subclasses of union_find_set are expected to override this default method if they want to take action when two equivalence classes are merged.

```
end module UnionFindSet;
```

In `dominant-union-find-set.diesel`:

`Dominant_union_find_set` is a subclass of `union_find_set` where its instances are known to always become the representative member of the equivalence class.

```
extend module Stdlib;

module DominantUnionFindSet;
abstract class dominant_union_find_set[T <= union_find_set[T]]
                isa union_find_set[T];

end module DominantUnionFindSet;
```

4.4 Tables (maps)

In `table.diesel`:

Tables map from keys to values (in other words, a table is a set of key/value pairs) such that a given key maps to at most one value. A table can be viewed as a collection of values, in some unspecified order. As such, operations like `length`, `is_empty`, `do`, `pick_any`, `includes`, `find`, `copy`, etc., are inherited from `collection`, and operate on the values part of the table.

A `table` is `comparable` if both its keys and values are `comparable`. Two tables are equal (`=`) if they have the same set of keys and corresponding keys map to equal values.

```
extend module Stdlib;

module Table;
```

first, a simple abstraction that supports `fetch`-like behavior, but not necessarily iterating, etc.

```
abstract class table_like[Key,Value]
                                -- Value type parameter is covariant:
                                isa table_like[Key, ‘Value1 >= Value];
abstract class table_like_exactly[Key,Value] isa table_like[Key,Value];
```

The `fetch` methods support table lookup; the optional closure argument is invoked if the key isn't found. The infix `!` operator can be used instead of `fetch`, e.g.:

```
   t!n1 + t!n2

  fun fetch(:table_like[‘Key,‘Value], key:Key,
                 if_absent:&():‘Else):Value|Else;
  fun fetch(t:table_like[‘Key,‘Value], key:Key):Value;
  fun !(t:table_like[‘Key,‘Value], key:Key):Value;
```

The `fetch_and_do` method does a fetch on the key and invokes either the if_present closure on the corresponding value or the if_absent closure if the key isn't found.

```
  fun fetch_and_do(t:table_like[‘Key,‘Value], key:Key,
                        if_present:&(Value):‘T1, if_absent:&():‘T2):T1|T2;
```

now, the main table abstraction

```
abstract class table[Key, Value] isa collection[Value],
                                            table_like[Key, Value],
                                            -- Value type param is covariant:
                                            table[Key, ‘Value1 >= Value];
abstract class table_exactly[Key, Value]
                                        isa collection_exactly[Value],
                                            table_like_exactly[Key, Value],
                                            table[Key, Value];
```

The control structure `do_associations[_allowing_updates]` forms the heart of a table, iterating through the keys and values of the table in pairs. The `keys_do[_allowing_updates]` methods iterate through just the keys.

```
fun do_associations(:table[‘Key,‘Value], :&(Key,Value):void):void;
fun do_associations_allowing_updates(t:table[‘Key,‘Value],
                                            c:&(Key,Value):void):void;
```

The `do_associations_exit` et al. control structures are like the `do_exit` et al. control structures in that they allow premature exiting and continuing of an iteration.

```
fun do_associations_exit(t:table[‘Key,‘Value],
                                closure:&(Key, Value, &():none):void):void;
fun do_associations_exit_continue(t:table[‘Key,‘Value],
                                         closure:&(Key, Value,
                                                 &():none, &():none):void
                                         ):void;
fun do_associations_continue(t:table[‘Key,‘Value],
                                    closure:&(Key, Value, &():none):void
                                    ):void;
fun keys_do(t:table[‘Key,‘Value], c:&(Key):void):void;
fun keys_do_allowing_updates(t:table[‘Key,‘Value], c:&(Key):void):void;
fun copy_mutable(:table_exactly[‘Key,‘Value]):m_table[‘Key,Value];
```

The `find_key` method does reverse table lookup: given a value, find a key that maps to that value. The `includes_key` method tests whether a key is defined, and the `keys[_{set,list}]` operations return the collection of keys in the table.

```
fun find_key(t:table_exactly[‘Key,‘Value], value:Value):Key
                                                where =(:Value,:Value):bool;
fun find_key(t:table_exactly[‘Key,‘Value],
                    value:Value, if_absent:&():‘Else):Key|Else
                                                where =(:Value,:Value):bool;
fun find_key_generically(t:table[‘Key,‘Value], value:Value):Key
                                                where =(:Value,:Value):bool;
fun find_key_generically(t:table[‘Key,‘Value], value:Value,
                                if_absent:&():‘Else):Key|Else
                                                where =(:Value,:Value):bool;
fun pick_any_key(t:table[‘Key,‘Value]):Key;
fun pick_any_key(t:table[‘Key,‘Value], if_empty:&():‘Else):Key|Else;
fun includes_key(t:table[‘Key,‘Value], key:Key):bool;
fun keys(t:table[‘Key,‘Value]):collection_exactly[Key];
fun keys_set(t:table[‘Key <= comparable[Key],‘Value]):set[Key];
fun keys_list(t:table[‘Key,‘Value]):ordered_collection_exactly[Key];
extend class table_exactly[‘Key, ‘Value <= comparable[Value]]
                                isa comparable[table_exactly[Key,Value]];
fun values_print_string(t:table[‘Key,‘Value]):string;
```

Tables are refined into immutable and mutable varieties. An `i_table` is immutable.

```
abstract class i_table[Key, Value] isa table[Key, Value],
                                             -- type param is covariant:
                                             i_table[Key, ‘Value1 >= Value];
abstract class i_table_exactly[Key, Value]
                                               isa table_exactly[Key, Value],
                                                   i_table[Key, Value];
```

a simple thing that supports `fetch` & `store`, but not necessarily iterating

```
abstract class m_table_like[Key,Value]
                                        isa table_like_exactly[Key,Value];
  fun store(:m_table_like[‘Key,‘Value], :Key, :Value):void;
  fun set_!(t:m_table_like[‘Key,‘Value], key:Key, value:Value):void;
```

An `m_table` supports changing bindings of keys to values through the `store` or `set_!` method, but not necessarily adding new keys or removing old ones. (See `removable_table` for operations for removing keys from tables.)

```
abstract class m_table[Key, Value] isa table_exactly[Key, Value],
                                           m_table_like[Key, Value];
fun store(:m_table[‘Key,‘Value], key:Key, value:Value,
                if_absent:&():void):void;
fun store_no_dup(t:m_table[‘Key,‘Value], key:Key, value:Value):void;
fun store(t:m_table[‘Key <= comparable[Key],‘Value],
                assocs:collection[assoc[Key,Value]]):void;
```

An `m_table` also supports a variation of `fetch`, `fetch_or_init`, that, if the key is not found, computes and adds a default value to the table and returns that value; `fetch_or_init` abstracts a very common table-manipulation idiom.

```
fun fetch_or_init(t:m_table[‘Key,‘Value], key:Key,
                        if_init:&():Value):Value;
```

`store_all` stores all the key/value bindings of the second table into the first table

```
fun store_all(t1:m_table[‘Key,‘Value], t2:table[‘Key,‘Value]):void;
```

`replace_any` replaces some occurrence of a value in a table with some other value

```
fun replace_any(t:m_table[‘Key,‘Value <= comparable[Value]],
                      old_value:Value, new_value:Value):void;
fun replace_any(t:m_table[‘Key,‘Value <= comparable[Value]],
                      old_value:Value, new_value:Value,
                      if_absent:&():void):void;
```

`replace_all` replaces all occurrences of a value in a table with some other value, returning the number of replacements made

```
fun replace_all(t:m_table[‘Key,‘Value],
                      old_value:Value, new_value:Value):int
        where signature =(:Value,:Value):bool;
```

`replace_if` replaces all occurrences of a value in a table that passes a predicate with some other value computed by a different function, returning the number of replacements made

```
fun replace_if(t:m_table[‘Key,‘Value],
                     pred:&(Value):bool, new_value_cl:&(Value):Value):int;
```

A `removable_table` supports removing bindings from the table, given the key to remove. (A table that inherits from `removable_collection`, on the other hand, supports removing bindings from the table, given the value.)

```
abstract class removable_table[Key, Value] isa table_exactly[Key,Value];
fun remove_key(:removable_table[‘Key,‘Value], key:Key,
                     if_absent:&():‘Else):Value|Else;
fun remove_key(t:removable_table[‘Key,‘Value], key:Key):Value;
fun remove_keys_if(t:removable_table[‘Key,‘Value],
                         pred:&(Key):bool):int;
```

`m_removable_table` is the commonly used kind of table, which supports addition, removal, and modification of key-to-value bindings. Both addition and modification of bindings is done via `store` or `set_!`. (For non-table collections, addition of elements is done via `add` methods.) The `set_!` method can be invoked using the assignment message sugar:

```
t ! key := value;
```

Removal of bindings is done via the inherited `remove_key` et al. methods.

```
abstract class m_removable_table[Key, Value]
        isa m_table[Key, Value], removable_table[Key, Value];

end module Table;
```

4.4.1 Concrete implementations

Several concrete implementations of tables exist: `assoc_table` is based on a linked-list of key/value associations (where the keys must be comparable), `identity_assoc_table` is like `assoc_table` but uses object identity (`==`) to compare keys, `hash_table` uses an open hashing algorithm, and `chained_hash_table` uses a closed hashing algorithm; the hashing versions require keys to be `hashable`.

The {`assoc`,`identity_assoc`,`hash`, `chained_hash`}`_CR_table` implementations change the printing behavior to print a newline between elements; this is useful for large tables. The "`_CR`" stands for "carriage return".

In `assoc-table.diesel`:

`assoc_table` is an implementation of tables based on a linked-list of key/value associations. The keys must be `comparable`.

```
extend module Stdlib;

module Assoc;
class assoc[Key <= comparable[Key], Value];
  field key(:assoc[‘Key,‘Value]):Key;
  var field value(:assoc[‘Key,‘Value]):Value;
  extend class assoc[‘Key, ‘Value <= comparable[Value]]
                                        isa comparable[assoc[Key,Value]];
  fun new_assoc[Key <= comparable[Key], Value](k:Key, v:Value
                                                     ):assoc[Key,Value];
  fun ==>(k:‘Key <= comparable[Key], v:‘Value):assoc[Key,Value];


end module Assoc;

module AssocTable;
let var warn_for_assoc_tables_longer_than:int;
class assoc_table[Key <= comparable[Key], Value]
                                        isa m_removable_table[Key,Value];
  fun new_assoc_table[Key <= comparable[Key], Value](
        ):assoc_table[Key,Value];
  fun new_assoc_table_init_from(
        assocs:collection[assoc[‘Key <= comparable[Key], ‘Value]]
        ):assoc_table[Key,Value];
class assoc_CR_table[Key <= comparable[Key], Value]
                                                isa assoc_table[Key,Value];
  fun new_assoc_CR_table[Key <= comparable[Key], Value](
        ):assoc_CR_table[Key,Value];
class ordered_assoc_table[Key <= comparable[Key], Value]
                                        isa assoc_table[Key,Value];
  fun new_ordered_assoc_table[Key <= comparable[Key], Value](
        ):ordered_assoc_table[Key,Value];
class ordered_assoc_CR_table[Key <= comparable[Key], Value]
                        isa ordered_assoc_table[Key,Value];
  fun new_ordered_assoc_CR_table[Key <= comparable[Key], Value](
        ):ordered_assoc_CR_table[Key,Value];

end module AssocTable;
```

In `identity-table.diesel`:

`identity_assoc_table` is an implementation of tables based on a linked-list of key/value associations. It uses object identity (`==`) to compare keys.

```
extend module Stdlib;
```

```
module IdentityAssoc;
class identity_assoc[Key,Value]
                                isa comparable[identity_assoc[Key,Value]];
  field key(:identity_assoc[‘Key,‘Value]):Key;
  var field value(:identity_assoc[‘Key,‘Value]):Value;
  fun new_identity_assoc[Key,Value](k:Key, v:Value
                                           ):identity_assoc[Key,Value];

end module IdentityAssoc;

module IdentityTable;
class identity_assoc_table[Key,Value]
                isa m_removable_table[Key,Value];
  fun new_identity_assoc_table[Key,Value]()
                                        :identity_assoc_table[Key,Value];
class identity_assoc_CR_table[Key,Value]
                        isa identity_assoc_table[Key,Value];
  fun new_identity_assoc_CR_table[Key,Value]()
                           :identity_assoc_CR_table[Key,Value];

end module IdentityTable;
```

In `small-table.diesel`:

An implementation of `m_removable_table` that is space efficient when there are only a few elements, but scales well when there are a large number of elements. Keys of small tables are required to be `hashable`.

```
extend module Stdlib;

module SmallTable;
class small_table[K <= hashable[K], V] isa m_removable_table[K,V];
fun shrink_table(t:small_table[‘K,‘V]):void;
fun new_small_table[K <= hashable[K], V]():small_table[K,V];
module AbsentTable;
synonym big_table[‘K <= hashable[K],‘V] =
  m_removable_table[K,V]|absent_table[K,V];
object absent_table[K <= hashable[K], V];

end module AbsentTable;

end module SmallTable;
```

Hash tables The hash table constructors optionally take a size argument that is a non-binding guess as to the maximum size of the table, used to pre-allocate a reasonable amount of hash vector space. All hash tables automatically resize if the table grows too large or small.

In `hash-table.diesel`:

```
extend module Stdlib;

module OpenHashTableImpl;
fun next_probe(v:vector[‘T], idx:int):int;
fun previous_probe(v:vector[‘T], idx:int):int;
fun buckets_in_quadratic_probing_order_do(v:vector[‘T], start_idx:int,
                                                 cl:&(int,T):void
                                                 ):void;
abstract class open_table[Key];
fun probe_histogram(t:open_table[‘Key <= hashable[Key]]
                           ):histogram[int];
```

```
end module OpenHashTableImpl;
```

```
module HashTable;
```

hash_table is an implementation of tables that uses an open hashing algorithm, for hashable keys.

```
class hash_table[Key <= hashable[Key], Value]
                        isa m_removable_table[Key,Value];
fun check_correctness(t:hash_table[‘Key,‘Value]):void;
fun new_hash_table[Key <= hashable[Key], Value]()
                                                    :hash_table[Key,Value];
fun new_hash_table[Key <= hashable[Key], Value](size:int)
                                                    :hash_table[Key,Value];
class hash_CR_table[Key <= hashable[Key], Value]
                                                isa hash_table[Key,Value];
fun new_hash_CR_table[Key <= hashable[Key], Value]()
                                               :hash_CR_table[Key,Value];
fun new_hash_CR_table[Key <= hashable[Key], Value](size:int)
                                               :hash_CR_table[Key,Value];
```

```
end module HashTable;
```

In chained-hash-table.diesel:
chained_hash_table is an implementation of tables using a closed hashing algorithm.

```
extend module Stdlib;
```

```
module ChainedHashTable;
class chained_hash_table[Key <= hashable[Key], Value]
                                        isa m_removable_table[Key,Value];
fun new_chained_hash_table[Key <= hashable[Key], Value](
      ):chained_hash_table[Key,Value];
fun new_chained_hash_table[Key <= hashable[Key], Value](
      size:int):chained_hash_table[Key,Value];
fun probe_histogram(t:chained_hash_table[‘Key,‘Value]):histogram[int];
class chained_hash_CR_table[Key <= hashable[Key], Value]
        isa chained_hash_table[Key,Value];
fun new_chained_hash_CR_table[Key <= hashable[Key], Value](
      ):chained_hash_CR_table[Key,Value];
fun new_chained_hash_CR_table[Key <= hashable[Key], Value](
      size:int):chained_hash_CR_table[Key,Value];
```

```
end module ChainedHashTable;
```

4.5 Ordered collections and sequences

In ordered.diesel:
The elements of an ordered collection appear in some well-defined order, i.e., do returns the elements in the same order each time called (specific subclasses define how that order is determined). Since all the classes in this section inherit from collection, they support length, is_empty, do, pick_any, includes, find, copy, and other collection operations.

```
extend module Stdlib;
```

```
module OrderedCollection;
abstract class ordered_collection[T] isa collection[T],
                                                -- type param is covariant:
                                                ordered_collection[‘S >= T];
abstract class ordered_collection_exactly[T] isa collection_exactly[T],
                                                        ordered_collection[T];
```

One through four ordered collections can be iterated through in parallel using `do`; the iteration exits when the smallest collection is exhausted.

```
fun do(c1:ordered_collection[‘T1], c2:ordered_collection[‘T2],
             c:&(T1,T2):void):void;
fun do(c1:ordered_collection[‘T1], c2:ordered_collection[‘T2],
             c3:ordered_collection[‘T3], c:&(T1,T2,T3):void):void;
fun do(c1:ordered_collection[‘T1], c2:ordered_collection[‘T2],
             c3:ordered_collection[‘T3], c4:ordered_collection[‘T4],
             c:&(T1,T2,T3,T4):void):void;
```

An ordered collection can be iterated through in reverse order, and a sequence can be constructed from the ordered collection in reverse order.

```
fun reverse(c:ordered_collection[‘T]):sequence[T];
fun reverse_do(c:ordered_collection[‘T], cl:&(T):void):void;
fun reverse_do(c1:ordered_collection[‘T1], c2:ordered_collection[‘T2],
                     cl:&(T1,T2):void):void;
fun reverse_do(c1:ordered_collection[‘T1], c2:ordered_collection[‘T2],
                     c3:ordered_collection[‘T3], cl:&(T1,T2,T3):void):void;
fun reverse_do(c1:ordered_collection[‘T1], c2:ordered_collection[‘T2],
                     c3:ordered_collection[‘T3], c4:ordered_collection[‘T4],
                     cl:&(T1,T2,T3,T4):void):void;
```

Ordered collections of comparable, partially-ordered, totally-ordered, or hashable elements are themselves comparable, partially-ordered, totally-ordered, or hashable, respectively. Two ordered collections are equal (`=`) if they have equal elements in the same order. Lexicographic (dictionary) ordering is used to compare two collections.

allow `=` etc. on any ordered collection, even inexactly known ones

```
extend class ordered_collection[‘T <= comparable[T]]
                isa comparable[ordered_collection[T]];
```

allow `hash` on any ordered collection, even inexactly known ones

```
extend class ordered_collection[‘T <= hashable[T]]
                isa hashable[ordered_collection[T]];
let hash_shift:int;
let num_hash_bits:int;
extend class ordered_collection_exactly[‘T <= ordered[T]]
                isa ordered_using_compare[ordered_collection_exactly[T]];
extend class ordered_collection_exactly[‘T <= ordered_hashable[T]]
                isa ordered_hashable[ordered_collection_exactly[T]];
fun as_ordered_collection(c:collection[‘T]):ordered_collection[T];
fun view_stream(:ordered_collection[‘T]):stream[T];
```

Elements may be extracted from ordered collections; however, these operations may be inefficient, particularly `fetch`, `next_to_last`, and `last`. Objects of type `indexed` afford fast access to any element.

```
fun first (a:ordered_collection[‘T]):T;
fun second(a:ordered_collection[‘T]):T;
fun third (a:ordered_collection[‘T]):T;
fun fourth(a:ordered_collection[‘T]):T;
fun next_to_last(a:ordered_collection[‘T]):T;
fun last  (a:ordered_collection[‘T]):T;
```

Ordered collections of strings can be flattened into a single string, optionally inserting a separator string element strings (only between non-empty element strings, for `flatten_ignoring_empty`). Similarly, any ordered collection can be flattened into a string using a user-defined closure to convert from the collection element to a string element.

```
fun flatten(c:ordered_collection[string]):string;
fun flatten(c:ordered_collection[string], sep:string):string;
fun flatten_ignoring_empty(c:ordered_collection[string],
                                  sep:string):string;
fun flatten_eval(c:collection[`T], cl:&(T):string):string;
fun flatten_eval(c:collection[`T], sep:string, cl:&(T):string):string;
fun flatten_eval_ignoring_empty(c:collection[`T], sep:string,
                                       cl:&(T):string):string;
```

Similarly, ordered collections of sequences can be flattened into a single vector

```
fun flat_vector(c:ordered_collection[`T <= sequence_exactly[`S]]
                      ):vector[S];
fun elems_print_string(c:collection[`T], sep:string):string;

end module OrderedCollection;
```

Extensible ordered collections can have their first and last elements removed, as well as the removing and adding behavior of extensible collections.

```
module ExtensibleOrderedCollection;
abstract class extensible_ordered_collection[T]
                isa extensible_collection[T], ordered_collection_exactly[T];
fun remove_first(:extensible_ordered_collection[`T],
                        if_empty:&():`S):T|S;
fun remove_first(c:extensible_ordered_collection[`T]):T;
fun remove_last(:extensible_ordered_collection[`T],
                       if_empty:&():`S):T|S;
fun remove_last(c:extensible_ordered_collection[`T]):T;
fun remove_last_N(c:extensible_ordered_collection[`T], n:int):void;  -- remove the last N elements

end module ExtensibleOrderedCollection;
```

In `sequence.diesel`:

A `sequence` is an ordered collection where the ordering of the elements is determined externally, by the way they were put into the collection. (In contrast, a sorted collection is an ordered collection where the order is determined by the elements themselves and the `<` binary ordering predicate over the elements.)

```
extend module Stdlib;

module Sequence;
abstract class sequence[T] isa ordered_collection[T],
                                      -- type param is covariant:
                                      sequence[`S >= T];
abstract class sequence_exactly[T] isa ordered_collection_exactly[T],
                                              sequence[T];
```

Concatenating two sequences always produces a new sequence

```
fun ||(s1:sequence[`T1], s2:sequence[`T2]):sequence[T1|T2];
```

```
end module Sequence;
```

Extensible sequences also allow elements to be added to the front or end of the sequence.

```
module ExtensibleSequence;
abstract class extensible_sequence[T]
                isa extensible_ordered_collection[T], sequence_exactly[T];
fun add_first(:extensible_sequence[`T], x:T):void;
fun add_last (:extensible_sequence[`T], x:T):void;
fun add_all_last(s:extensible_sequence[`T], c:collection[T]):void;

end module ExtensibleSequence;
```

4.6 Indexed collections: vector, array, string, list, ...

In `indexed.diesel`:

Indexed collections are indexed sequences. They support the behavior of keyed tables, where the integers from 0 to c.length-1 serve as the keys. Indexed collections inherit: `length`, `is_empty`, `do`, `pick_any`, `includes`, `find`, `copy`, etc., from collections; `reverse_do`, `flatten`, `||`, etc., from sequences; `fetch`, `!`, `do_associations`, `keys_do`, etc., from tables.

```
extend module Stdlib;

module IndexedCollection;
abstract class indexed[T] isa sequence[T], table[int,T],
                                     -- type parameter is covariant:
                                     indexed[`S >= T];
abstract class indexed_exactly[T] isa sequence_exactly[T],
                                             table_exactly[int,T],
                                             indexed[T];
```

`range_do` evaluates c on all elements in [max(start,0)..min(stop,len)) (note half-open interval!)

```
fun range_do(a:indexed[`T], start:int, stop:int,
                   c:&(int,T):void):void;
```

allow `includes` etc. on any indexed collection, even inexactly known ones

The `includes_index` and `find_index` methods are simply renamings of the standard `includes_key` and `find_key` methods.

```
fun includes_index(a:indexed[`T], i:int):bool;
```

allow `find_index` on any indexed collection, even inexactly known ones

```
fun find_index(a:indexed[`T], value:T):int where =(:T,:T):bool;
fun find_index(a:indexed[`T], value:T,
                     if_absent:&():`S):int|S where =(:T,:T):bool;
```

The `copy_from` method copies a portion of an indexed collection from a start index up to (but not including) a stop index (or the end of the collection, if the stop index is not specified). The `has_{prefix,suffix}` functions test whether an indexed collection starts with or ends with a particular collection; the `remove_{prefix,suffix}` functions return a new indexed collection with the specified prefix or suffix removed, if present, or the original indexed collection otherwise. New collections use the `factory` framework to try to return a collection of the same kind as the argument.

```
fun copy_from(s:‘S <= indexed[‘T], start:int):S
                                where copy_from(:S, :int, :int):S;
fun copy_from(s:indexed[‘T], start:int, up_to:int):indexed[T];
fun has_prefix(s:indexed[‘T <= comparable[T]],
                     prefix:indexed[‘T]):bool;
fun has_suffix(s:indexed[‘T <= comparable[T]],
                     suffix:indexed[‘T]):bool;
fun remove_prefix(s:‘S <= indexed[‘T <= comparable[T]],
                        prefix:indexed[‘T]):S
        where copy_from(:S, :int, :int):S;
fun remove_suffix(s:‘S <= indexed[‘T <= comparable[T]],
                        suffix:indexed[‘T]):S
        where copy_from(:S, :int, :int):S;
```

The `split` function breaks up an indexed collection into segments based on a separator element. Some segments may be empty if the separator is at the beginning, end, or adjacent to another separator.

```
fun split(s:‘S <= indexed[‘T],
                 separator:‘T <= comparable[T]
                 ):extensible_sequence[S]
        where copy_from(:S, :int, :int):S;
```

As usual, there are immutable and mutable varieties of indexed collections. Mutable collections support changing bindings of indices to values through the `store` or `set_!` methods inherited from `m_table`.

```
abstract class i_indexed[T] isa indexed[T], i_table[int,T],
                                       -- type param is covariant:
                                       i_indexed[‘S >= T];
abstract class i_indexed_exactly[T] isa indexed_exactly[T],
                                               i_table_exactly[int,T],
                                               i_indexed[T];
abstract class m_indexed[T] isa indexed_exactly[T], m_table[int,T];
fun as_m_indexed(c:collection_exactly[‘T]):m_indexed[T];
```

There are convenient ways of assigning into the beginning and end of a collection. They can be invoked using the assignment message sugar, e.g., `first(c) := x`.

```
fun set_first (a:m_indexed[‘T], x:T):void;
fun set_second(a:m_indexed[‘T], x:T):void;
fun set_third (a:m_indexed[‘T], x:T):void;
fun set_fourth(a:m_indexed[‘T], x:T):void;
fun set_last  (a:m_indexed[‘T], x:T):void;
fun set_only(a:m_indexed[‘T], x:T):void;
```

The swap method exchanges the indices of two collection elements.

```
fun swap(c:m_indexed[‘T], i:int, j:int):void;
```

The `slide_elems[_by](from,to[,amount])` methods copy elements in the range `[from..to]`, inclusive, up by `amount` positions (where `amount` defaults to 1), sliding down if `amount` is negative.

```
fun slide_elems(t:m_indexed[‘T], from:int, to:int):void;
fun slide_elems_by(t:m_indexed[‘T], from:int, to:int, amount:int):void;
```

The `sort` method sorts a collection in place, using either the element type’s natural comparison operation (`<=`) or a user-supplied comparator function. Sorting uses the quicksort algorithm and attempts to be a reasonably stable sort.

```
fun sort(c:m_indexed[‘T <= ordered[T]]):void;
fun sort_by(c:m_indexed[‘T], pred:&(T,T):bool):void;
fun sort_by(c:m_indexed[‘T], pred:&(T,T):bool,
                   first_index:int, last_index:int):void;
```

The `pos` method returns the index at which the first collection occurs in the second collection (invoking the `if_absent` closure if not found), using the Knuth-Morris-Pratt string-matching algorithm. The `has_subsequence` method returns whether the second collection is found in the first collection, and the `count_subsequences` method returns the number of non-overlapping occurrences of the second collection in the first. [Note that the order of collection arguments in `pos` is opposite to that in the `subsequence` methods!]

```
fun pos(s1:indexed_exactly[‘T <= comparable[T]],
               s2:indexed_exactly[T]):int;
fun pos(s1:indexed_exactly[‘T <= comparable[T]],
               s2:indexed_exactly[T],
               if_fail:&():int):int;
fun has_subsequence(s1:indexed_exactly[‘T <= comparable[T]],
                           s2:indexed_exactly[T]):bool;
fun count_subsequences(s1:indexed_exactly[‘T <= comparable[T]],
                              s2:indexed_exactly[T]):int;
```

An `indexed_factory` is the abstract superclass of factories for indexed collections.

```
abstract class indexed_factory;
```

`generic_factory` returns a factory to use, if the receiver factory doesn't implement some operation

```
  fun generic_factory(:indexed_factory):indexed_factory;
```

`mutable_factory` returns the factory to use if a mutable result is required

```
  fun mutable_factory(:indexed_factory):m_indexed_factory;
```

`create_empty` yields a zero-length indexed collection of the right kind.

```
  fun create_empty[T](f:indexed_factory):indexed_exactly[T];
```

`create_filled` yields a new indexed collection of the right kind, with the given length, all of whose elements are `filler`.

```
  fun create_filled(f:indexed_factory,
                           size:int, filler:‘T):indexed[T];
  fun create_filled[T](f:indexed_factory,
                              size:int, filler:T):indexed_exactly[T];
```

`create_init` yields a new indexed collection of the right kind, with the given length, whose i'th element is the result of evaluating the closure on i.

```
  fun create_init(f:indexed_factory,
                         size:int, cl:&(int):‘T):indexed[T];
  fun create_init[T](f:indexed_factory,
                            size:int, cl:&(int):T):indexed_exactly[T];
```

`create_mapped` yields a new indexed collection of the right kind, whose length is the same as the argument collection's and whose i'th element is the result of evaluating the closure on the i'th element of the argument collection.

```
  fun create_mapped(f:indexed_factory,
                           c:collection[‘T1], cl:&(T1):‘T2):indexed[T2];
  fun create_mapped[T2](f:indexed_factory,
                               c:collection[‘T1], cl:&(T1):T2
                               ):indexed_exactly[T2];
```

Like create_mapped, except that the argument closure is invoked on each key/value pair of the argument collection.

```
  fun create_mapped_associations(f:indexed_factory,
                                        c:table[‘K,‘T1], cl:&(K,T1):‘T2
                                        ):indexed[T2];
  fun create_mapped_associations[T2](f:indexed_factory,
                                            c:table[‘K,‘T1], cl:&(K,T1):T2
                                            ):indexed_exactly[T2];
```

create_copy returns a new indexed collection of the right kind whose length and elements are the same as the argument collection's.

```
  fun create_copy(f:indexed_factory, c:collection[‘T]):indexed[T];
  fun create_copy[T](f:‘F <= indexed_factory, c:collection[T]):‘C
        where signature create_mapped[T](:F,:collection[T],:&(T):T):C;
abstract class i_indexed_factory isa indexed_factory;
abstract class m_indexed_factory isa indexed_factory;
```

create_sized yields a new indexed collection of the right kind, with the given length, all of whose elements are initialized with some default "uninitialized" value.

```
  fun create_sized[T](f:m_indexed_factory, size:int):m_indexed[T];
```

factory returns the factory object for the argument collection.

```
fun factory(:indexed[‘T]):indexed_factory;
fun map(t:indexed[‘T1], cl:&(T1):‘T2):indexed[T2];  -- a traditional map function
fun map_associations(t:indexed[‘T1], cl:&(int,T1):‘T2):indexed[T2];  -- a version that maps over index/value pairs

end module IndexedCollection;
```

4.6.1 Implementations

Several concrete implementations of indexed collections exist:

- i_vectors are primitive fixed-length immutable indexed collections, item m_vectors are primitive fixed-length mutable indexed collections,
- arrays are extensible mutable indexed collections,
- intervals are immutable indexed collections representing a finite arithmetic sequence of integers,
- strings are indexed collections of characters, with mutable and immutable varieties,
- bit_vectors are fixed-length mutable indexed collections of 0/1 values,
- word_vectors are fixed-length mutable indexed collections of single-word short integers, and
- float_vectors are fixed-length mutable indexed collections of floating-point values (short integers).

In vector.diesel:

i_vector is a primitive implementation of fixed-length immutable indexed collections. Besides the new_ methods (see the next paragraph on how they work), instances of i_vector can be created by Diesel vector constructor expressions, e.g., [3, 4, 5, x*12].

m_vector is a primitive implementation of fixed-length mutable indexed collections.

The new_(i|m)_vector function constructs a new (i|m)_vector of a given size all of whose elements are filler. The new_(i|m)_vector_init function constructs a (i|m)_vector of a particular size and uses the elems closure to construct the initial values of the vector's elements from the indices. The new_(i|m)_vector_init_from function constructs a (i|m)_vector of the same length as some other ordered collection and applies a mapping function to translate each element of the receiver collection into the corresponding vector element; new_(i|m)_vector_init_from is simply the well-known map function, specialized to return a particular vector representation. In the current implementation, vectors cannot be subclassed.

```
extend module Stdlib;

module Vector;
abstract class vector[‘T] isa indexed[T],
                              -- type parameter is covariant:
                              vector[‘S >= T];
abstract class vector_exactly[‘T] isa indexed_exactly[T], vector[T];
fun as_vector(c:collection[‘T]):vector[T];
```

A vector_factory is the abstract superclass of factories for vectors.

```
abstract class vector_factory isa indexed_factory;
abstract class i_vector[T] isa vector[T], i_indexed[T],
                               i_vector[‘S >= T];
abstract class i_vector_exactly[T] isa vector_exactly[T],
                                       i_indexed_exactly[T],
                                       i_vector[T];
fun new_i_vector(size:int, filler:‘T):i_vector_exactly[T];
fun new_i_vector[T](size:int, filler:T):i_vector_exactly[T];
fun new_i_vector_init[T](size:int, cl:&(int):T):i_vector_exactly[T];
fun new_i_vector_from(c:collection[‘T1],
                      cl:&(T1):‘T2):i_vector_exactly[T2];
fun new_i_vector_init_from[T2](c:collection[‘T1],
                               cl:&(T1):T2):i_vector_exactly[T2];
fun new_i_vector_from_associations(c:table[‘K,‘T1],
                                   cl:&(K,T1):‘T2
                                   ):i_vector_exactly[T2];
fun new_i_vector_init_from_associations[T2](c:table[‘K,‘T1],
                                            cl:&(K,T1):T2
                                            ):i_vector_exactly[T2];
fun as_i_vector(c:collection[‘T]):i_vector[T];
fun as_i_vector[T](c:collection[T]):i_vector_exactly[T];
```

An i_vector_factory is the concrete factory object for i_vectors.

```
object i_vector_factory isa vector_factory, i_indexed_factory;
abstract class m_vector[T] isa vector_exactly[T], m_indexed[T];
fun new_m_vector(size:int):m_vector[dynamic];
fun new_m_vector[T](size:int):m_vector[T];
fun new_m_vector(size:int, filler:‘T):m_vector[T];
fun new_m_vector[T](size:int, filler:T):m_vector[T];
fun new_m_vector_init[T](size:int, cl:&(int):T):m_vector[T];
fun new_m_vector_from(c:collection[‘T1], cl:&(T1):‘T2):m_vector[T2];
fun new_m_vector_init_from[T2](c:collection[‘T1],
                               cl:&(T1):T2):m_vector[T2];
fun new_m_vector_from_associations(c:table[‘K,‘T1],
```

```
                                                   cl:&(K,T1):`T2):m_vector[T2];
fun new_m_vector_init_from_associations[T2](c:table[`K,`T1],
                                                      cl:&(K,T1):T2
                                                      ):m_vector[T2];
```

resize returns a new m_vector new_size long whose first elements are copied from v. if new_size is greater than v.length, then the extra elements are initialized to filler.

```
fun resize(v:vector[`T], new_size:int, filler:T):m_vector[T];
fun as_m_vector(c:collection_exactly[`T]):m_vector[T];
fun as_m_vector[T](c:collection[T]):m_vector[T];
```

An m_vector_factory is the concrete factory object for m_vectors.

```
object m_vector_factory isa vector_factory, m_indexed_factory;

end module Vector;
```

In array.diesel:

arrays are extensible mutable indexed collections. Arrays support both indexing behavior (fetch, store, find_index, etc.) and extensible sequence behavior (add_first, add_last, remove_first, etc.). add adds to the end of the array.

The new_array() function returns a new empty array, as does the new_array(max_size) function (which additionally accepts a non-binding guess as to the default maximum size of the array). The other new_array functions are analogues of the corresponding new_*_vector functions, but return an array instead of a vector.

```
extend module Stdlib;

module Array;
class array[T] isa m_indexed[T], extensible_sequence[T];
extend class array[`T <= comparable[T]] isa removable_collection[T];
fun new_array[T]():array[T];
fun new_array[T](size:int):array[T];
fun new_array[T](size:int, filler:T):array[T];
fun new_array_init[T](size:int, cl:&(int):T):array[T];
fun new_array_init_from[T2](c:collection[`T1], cl:&(T1):T2):array[T2];
fun new_array_init_from_associations[T2](c:table[`K,`T1],
                                                cl:&(K,T1):T2):array[T2];
fun as_array(c:collection_exactly[`T]):array[T];
fun as_array[T](c:collection[T]):array[T];
fun new_array():array[dynamic];
fun new_array(size:int):array[dynamic];
fun new_array(size:int, filler:dynamic):array[dynamic];
fun remove_index(a:array[`T], index:int):T;
fun remove_index(a:array[`T], index:int, if_absent:&():`S):T|S;
fun quick_remove_all(a:array[`T]):void;

end module Array;
```

In interval.diesel:

intervals are immutable indexed collections representing a finite arithmetic sequence of integers. An interval returned by new_interval(start, stop) represents the integers in order from start to stop, inclusive; if stop is less than start then the sequence is empty. The interval may optionally specify a step value between start and stop. If the step is negative, then the sequence goes from start down to stop, inclusive; if stop is greater than start, then the sequence is empty.

```
extend module Stdlib;

module Interval;
class interval isa i_indexed_exactly[int];
  field start(:interval):int;
  field stop(:interval):int;
  field step(:interval):int;
fun new_interval(start:int, stop:int):interval;
fun new_interval(start:int, stop:int, step:int):interval;
```

Intervals can be used to implement regular for loops. For example,

```
for i := 1 to 10 do ... end
```

can be expressed in Diesel as:

```
new_interval(1, 10).do(&(i:int){
    ...
});
```

The `for` methods provide a convenient interface for this idiom; they also currently implement this idiom more efficiently. (Recently, compiler optimizations have been implemented that greatly reduce the performance cost of the new_interval(...).do(...) version; if debug_support is disabled, there should be no difference in performance between new_interval(...).do(...) and for(...).)

```
fun for(start:int, stop:int, body:&(int):void):void;
fun for(start:int, stop:int, step:int, body:&(int):void):void;

end module Interval;
```

In `bit-vector.diesel`:

A bit vector is a dense representation of a mutable indexed collection of zeros and ones. A new bit vector of zeros can be created with `new_bit_vector`. A bit vector can be resized in place; if expanded, then the new positions are filled in with zeros.

```
extend module Stdlib;

module BitVector;
class bit_vector isa m_indexed[int];
  field method length(@bit_vector):int;
fun new_bit_vector(sz:int):bit_vector;
fun resize(t:bit_vector, new_num_bits:int):bit_vector;
fun =_or_zero(l:bit_vector, r:bit_vector):bool;
```

Bit vectors can be or'd, and'd, xor'd, subtracted, and negated, all bitwise, to compute new bit vectors from old. For binary operations, the shorter bit vector is zero-extended before being operated on. The `..._in_place` versions try to construct the new bit_vector by overwriting the first bit vector, but if the first bit vector is too short, it will construct and return a new bit vector

```
fun bit_and(l:bit_vector, r:bit_vector):bit_vector;
fun bit_and_in_place(l:bit_vector, r:bit_vector):bit_vector;
fun bit_or(l:bit_vector, r:bit_vector):bit_vector;
fun bit_or_in_place(l:bit_vector, r:bit_vector):bit_vector;
fun bit_xor(l:bit_vector, r:bit_vector):bit_vector;
fun bit_xor_in_place(l:bit_vector, r:bit_vector):bit_vector;
fun bit_xnor(l:bit_vector, r:bit_vector):bit_vector;
fun bit_xnor_in_place(l:bit_vector, r:bit_vector):bit_vector;
fun bit_not(l:bit_vector):bit_vector;
fun bit_not_in_place(l:bit_vector):bit_vector;
fun bit_difference(l:bit_vector, r:bit_vector):bit_vector;
fun bit_difference_in_place(l:bit_vector, r:bit_vector):bit_vector;
```

A bit vector can be mutated to be all zeros (`clear_all_bits`) or all ones (`set_all_bits`) or tested for those conditions (`is_all_zeros` and `is_all_ones`). The `is_disjoint_bits` method returns true if its arguments share no set bits; `is_disjoint_bits(a,b)` is equivalent to `is_all_zeros(bit_and(a,b))`. The `includes_all_bits` method returns true if its first argument includes all the 1 bits of its second argument; `includes_all_bits(a,b)` is equivalent to `=_or_zero(bit_and(a,b),b)`.

```
fun includes_all_bits(l:bit_vector, r:bit_vector):bool;
fun is_disjoint_bits(l:bit_vector, r:bit_vector):bool;
fun clear_all_bits(v:bit_vector):void;
fun set_all_bits(v:bit_vector):void;
fun is_all_zeros(v:bit_vector):bool;
fun is_all_ones(v:bit_vector):bool;
```

The `do_ones` control structure iterates over all the set positions in the bit vector; this operation runs faster than a regular `do` loop containing a test in each iteration.

```
fun do_ones(v:bit_vector, cl:&(int):void):void;

end module BitVector;
```

In `byte-vector.diesel`:

```
extend module Stdlib;

module ByteVector;
abstract class byte_vector isa indexed_exactly[int];
fun as_byte_vector(c:collection[int]):byte_vector;
fun new_i_byte_vector(size:int, filler_oop:int):i_byte_vector;
fun new_i_byte_vector_init(size:int, cl:&(int):int):i_byte_vector;
fun new_i_byte_vector_init_from(c:collection[`T],
                                       cl:&(T):int):i_byte_vector;
fun as_i_byte_vector(c:collection[int]):i_byte_vector;
fun new_m_byte_vector(size:int):m_byte_vector;
fun new_m_byte_vector(size:int, filler_oop:int):m_byte_vector;
fun new_m_byte_vector_init(size:int, cl:&(int):int):m_byte_vector;
fun new_m_byte_vector_init_from(c:collection[`T],
                                       cl:&(T):int):m_byte_vector;
fun as_m_byte_vector(c:collection[int]):m_byte_vector;

end module ByteVector;
```

In `short-vector.diesel`:

```
extend module Stdlib;

module ShortVector;
abstract class short_vector isa indexed_exactly[int];
fun as_short_vector(c:collection[int]):short_vector;
fun new_i_short_vector(size:int, filler_oop:int):i_short_vector;
fun new_i_short_vector_init(size:int, cl:&(int):int):i_short_vector;
fun new_i_short_vector_init_from(c:ordered_collection[`T],
                                       cl:&(T):int):i_short_vector;
fun as_i_short_vector(c:collection[int]):i_short_vector;
fun new_m_short_vector(size:int):m_short_vector;
fun new_m_short_vector(size:int, filler_oop:int):m_short_vector;
fun new_m_short_vector_init(size:int, cl:&(int):int):m_short_vector;
fun new_m_short_vector_init_from(c:ordered_collection[`T],
                                       cl:&(T):int):m_short_vector;
fun as_m_short_vector(c:collection[int]):m_short_vector;
```

```
end module ShortVector;
```

In word-vector.diesel:

```
extend module Stdlib;

module WordVector;
abstract class word_vector isa indexed_exactly[int];
fun as_word_vector(c:collection[int]):word_vector;
fun new_i_word_vector(size:int, filler_oop:int):i_word_vector;
fun new_i_word_vector_init(size:int, cl:&(int):int):i_word_vector;
fun new_i_word_vector_init_from(c:ordered_collection[`T],
                                   cl:&(T):int):i_word_vector;
fun as_i_word_vector(c:collection[int]):i_word_vector;
fun new_m_word_vector(size:int):m_word_vector;
fun new_m_word_vector(size:int, filler_oop:int):m_word_vector;
fun new_m_word_vector_init(size:int, cl:&(int):int):m_word_vector;
fun new_m_word_vector_init_from(c:ordered_collection[`T],
                                   cl:&(T):int):m_word_vector;
fun as_m_word_vector(c:collection[int]):m_word_vector;

end module WordVector;
```

In float-vector.diesel:

```
extend module Stdlib;

module FloatVector;
abstract class float_vector isa indexed_exactly[float];
fun as_float_vector(c:collection[float]):float_vector;
fun new_i_float_vector(size:int, filler_oop:float):i_float_vector;
fun new_i_float_vector_init(size:int, cl:&(int):float):i_float_vector;
fun new_i_float_vector_init_from(c:collection[`T],
                                    cl:&(T):float):i_float_vector;
fun as_i_float_vector(c:collection[float]):i_float_vector;
fun new_m_float_vector(size:int):m_float_vector;
fun new_m_float_vector(size:int, filler_oop:float):m_float_vector;
fun new_m_float_vector_init(size:int, cl:&(int):float):m_float_vector;
fun new_m_float_vector_init_from(c:collection[`T],
                                    cl:&(T):float):m_float_vector;
fun as_m_float_vector(c:collection[float]):m_float_vector;

end module FloatVector;
```

4.6.2 Strings

In string.diesel:

The string type is an abstract class representing an indexed sequence of characters. Methods operating on generic strings that both take and return strings, in most cases produce a new string as a result.

```
extend module Stdlib;

module String;
abstract class string isa indexed_exactly[char];
```

In addition to all the operations available on other indexed collections, characters can be concatenated onto the front or back of a string using the `||` infix operator. (The `||` operator can also concatenate two strings. This behavior is inherited from sequences.)

The `pad` functions add either blanks or a specified padding character to either the front or the back of the string to make it be of at least the specified length.

```
fun pad(s:string, len:int):string;  -- pads on the right
fun pad_right(s:string, len:int):string;
fun pad_right(s:string, len:int, padding:char):string;
fun pad_left(s:string, len:int):string;
fun pad_left(s:string, len:int, padding:char):string;
```

A `string_factory` is the abstract superclass of factories for strings. It introduces a number of char-specific creation operations, parallel to the ones that are parameterized by the kind of element to create.

```
abstract class string_factory isa indexed_factory;
  fun create_char_empty(:string_factory):string;
  fun create_char_filled(:string_factory,
                                size:int, filler:char):string;
  fun create_char_init(:string_factory,
                              size:int, cl:&(int):char):string;
  fun create_char_mapped(:string_factory,
                                c:collection[`T1], cl:&(T1):char):string;
  fun create_char_mapped_associations(:string_factory,
                                             c:table[`K,`T1], cl:&(K,T1):char
                                             ):string;
  fun create_char_copy(f:`F <= string_factory, c:collection[char]):`S
           where create_char_mapped(:F,:collection[char],:&(char):char):S;
```

As usual, there are immutable and mutable varieties of strings.

```
abstract class i_string isa string, i_indexed_exactly[char];
abstract class i_string_factory isa string_factory, i_indexed_factory;
abstract class m_string isa string, m_indexed[char];
  fun write_into_string_at_pos(s1:string, s2:m_string, pos:int):void;
abstract class m_string_factory isa string_factory, m_indexed_factory;
  fun create_char_sized(:m_string_factory, size:int):m_string;
```

The `vstring` class and its two concrete subclasses, `i_vstring` and `m_vstring`, provide a primitive fixed-length packed string implementation. Diesel string literals (e.g., `"hello"`) are instances of `i_vstring`. Various constructors for `vstring`s are provided, analogously to strings and arrays.

```
abstract class vstring isa string;
fun as_vstring(c:collection[char]):vstring;
```

A `vstring_factory` is the abstract superclass of factories for vstrings.

```
abstract class vstring_factory isa string_factory;
abstract class i_vstring isa i_string, vstring;
fun new_i_vstring(size:int):i_vstring;
fun new_i_vstring(size:int, filler_oop:char):i_vstring;
fun new_i_vstring_init(size:int, cl:&(int):char):i_vstring;
fun new_i_vstring_init_from(c:collection[`T], cl:&(T):char):i_vstring;
fun new_i_vstring_init_from_associations(c:table[`K,`T],
                                                cl:&(K,T):char):i_vstring;
fun as_i_vstring(c:collection[char]):i_vstring;
```

An `i_vstring_factory` is the concrete factory object for i_vstrings.

```
object i_vstring_factory isa vstring_factory, i_string_factory;
abstract class m_vstring isa m_string, vstring;
fun new_m_vstring(size:int):m_vstring;
fun new_m_vstring_no_init(size:int):m_vstring;
fun new_m_vstring(size:int, filler_oop:char):m_vstring;
fun new_m_vstring_init(size:int, cl:&(int):char):m_vstring;
fun new_m_vstring_init_from(c:collection[‘T], cl:&(T):char):m_vstring;
fun new_m_vstring_init_from_associations(c:table[‘K,‘T],
                                                 cl:&(K,T):char):m_vstring;
fun as_m_vstring(c:collection[char]):m_vstring;
```

An `m_vstring_factory` is the concrete factory object for m_vstrings.

```
object m_vstring_factory isa vstring_factory, m_string_factory;

end module String;
```

4.7 Lists

In `list.diesel`:

Lists are sequences based on a linked-list representation. General lists support `first` (a.k.a. `car`, `head`) and `rest` (a.k.a. `cdr`, `tail`) operations, plus all the standard sequence operations, like `length`, `is_empty`, `do`, `pick_any`, `includes`, `find`, `copy`, `reverse_do`, `flatten`, `||`, etc.

```
extend module Stdlib;

module List;
abstract class list[T] isa sequence[T],
                                  -- type parameter is covariant:
                                  list[‘S >= T];
abstract class list_exactly[T] isa sequence_exactly[T], list[T];
  fun rest(:list[‘T]):list[T];
predicate empty_list[T] isa list[T], empty_collection[T];
predicate non_empty_list[T] isa list[T], non_empty_collection[T];
```

Simple lists are mutable Lisp-style singly-linked lists with two representations, `nil` and `cons`. Method `cons` creates a new cons cell; object `nil[T]` can be used directly. Simple lists are mutable, in the sense that `set_first` and `set_rest` operations are defined (they produce run-time errors when invoked on nil). However, simple lists are not `extensible_sequences` because they do not support `add` in place. Instead, `cons` (a.k.a. `add_functional`) adds to the front of a simple list, returning a new list.

```
abstract class simple_list[T] isa list_exactly[T],
                                         functionally_extensible_collection[T];
  fun set_rest(:simple_list[‘T], :simple_list[T]):void;
  fun append(l1:simple_list[‘T], l2:simple_list[T]):simple_list[T];
object nil[T] isa simple_list[T];
class cons[T] isa simple_list[T];
  fun cons(hd:T, tl:simple_list[‘T]):cons[T];
```

To correct this problem, the `m_list` type defines a full-fledged mutable, extensible list data structure, implemented as a wrapper around a simple list. `m_list` inherits `add` and `remove` from `extensible_collection`. For historical reasons, `add` is defined to `add` to the front of the list. `m_list` also inherits `{add,remove}_{first,last}` from `extensible_sequence`. (This data structure definition doesn't follow the usual pattern: add adds to the front of the collection, not to the end; there is no i_list data type defined and the m_list data type is concrete rather than abstract.) In the future, it would be really nice to define a view of doubly-linked lists that treats them as a sequence of link nodes. Then lots of list splicing operations could be supported that aren't really supported through the existing generic m_list interface. E.g. removing an element from a list during an iteration through it requires two traversals if written in terms of the m_list interface. Doubly-linked and circular lists might also be useful data structures.

```
class m_list[T] isa list_exactly[T], extensible_sequence[T];
extend class m_list[‘T <= comparable[T]] isa removable_collection[T];
  fun remove_and_return_one(l:m_list[‘T], test:&(T):bool,
                                  if_absent:&():‘Else):T|Else;
  fun splice_onto_end(l1:m_list[‘T], l2:m_list[T]):m_list[T];
```

`replace_any` replaces some occurrence of a value in a list with some other value

```
  fun replace_any(l:m_list[‘T <= comparable[T]],
                        old_value:T, new_value:T):void;
  fun replace_any(l:m_list[‘T <= comparable[T]],
                        old_value:T, new_value:T,
                        if_absent:&():void):void;
```

`replace_all` replaces all occurrences of a value in a list with some other value, returning the number of replacements made

```
  fun replace_all(l:m_list[‘T], old_value:T, new_value:T):int
          where signature =(:T,:T):bool;
```

`replace_if` replaces all occurrences of a value in a list that passes a predicate with some other value computed by a different function, returning the number of replacements made

```
  fun replace_if(l:m_list[‘T],
                       pred:&(T):bool, new_value_cl:&(T):T):int;
  fun new_m_list[T]():m_list[T];

end module List;
```

4.8 Sorted collections

In `sorted.diesel`:

A sorted collection is an ordered collection of totally-ordered values but not a sequence; the order of the elements is determined internally by the elements themselves (and their `<` method), not externally by the client.

Sorted collections inherit `length`, `is_empty`, `do`, `pick_any`, `includes`, `find`, `copy`, `reverse_do`, `flatten`, etc., from ordered collections.

In addition, mutable sorted collections support the behavior of extensible ordered collections, such as `add`, `remove_first`, and `remove_last` (but not `add_{first,last}`).

Two concrete implementations of sorted collections exist, one based on binary trees and another based on skip-lists.

```
extend module Stdlib;

module SortedCollection;
abstract class sorted_collection[T] isa ordered_collection[T],
                                               -- type param is covariant:
                                               sorted_collection[‘S >= T];
abstract class sorted_collection_exactly[T]
                                        isa ordered_collection_exactly[T],
                                            sorted_collection[T];
abstract class m_sorted_collection[T]
        isa sorted_collection_exactly[T], extensible_ordered_collection[T];
```

Simple binary trees are an implementation of sorted collections based on binary trees. They are akin to simple lists in that they support an add operation, but not in place, and so they do not support the full protocol of `m_sorted_collections`. Like `m_lists`, therefore, `m_binary_trees` are defined as convenient wrappers around `simple_binary_trees`.

```
abstract class simple_binary_tree[T <= ordered[T]]
                                        isa sorted_collection_exactly[T];
object empty_tree[T <= ordered[T]] isa simple_binary_tree[T];
class tree_node[T <= ordered[T]] isa simple_binary_tree[T];
```

`m_binary_tree` is a mutable sorted collection implemented as a wrapper around `simple_binary_tree`. Warning: the current implementation of binary trees does not support any of the remove protocol expected of an `m_sorted_collection`. This should be implemented sometime. Also, these binary trees are not self-balancing, so in the worst case an add operation can take O(length) time.

```
class m_binary_tree[T <= ordered[T]] isa m_sorted_collection[T];
  fun new_sorted_collection[T <= ordered[T]]():m_binary_tree[T];

end module SortedCollection;
```

In `skiplist.diesel`:

Skip lists are an alternative implementation of mutable sorted collections that perform probabilistically better than balanced trees. Skip lists can be sorted using either the element type's natural ordering or using a user-supplied ordering function. Skip lists suppress duplicates, like sets.

```
extend module Stdlib;

module SkipList;
class skip_list[T <= comparable[T]] isa m_sorted_collection[T],
                                          removable_collection[T];
  fun new_skip_list[T <= partially_ordered[T]]():skip_list[T];
  fun new_predicate_skip_list[T <= comparable[T]](cmp:&(T,T):bool
                                                    ):skip_list[T];
module SkipListImpl;
abstract class skip_list_node[T];
  fun value(:skip_list_node['T]):T;
  fun is_nil(:skip_list_node['T]):bool;
class skip_list_cons[T] isa skip_list_node[T];
  field forward(:skip_list_cons['T]):array[skip_list_node[T]];
  fun new_skip_list_node['T](level:int, new_value:T):skip_list_cons[T];
object skip_list_nil[T] isa skip_list_node[T];
object skip_list_nil_value isa ordered['T];
object rand_sl_level_stream;
  field rs(:rand_sl_level_stream):random_stream;
  fun next(r:rand_sl_level_stream, current_level:int):int;

end module SkipListImpl;

end module SkipList;
```

4.9 stack, queue

Stacks and queues are implemented as special interfaces to mutable lists.

In `stack.diesel`:

Stacks are implemented as special interfaces to mutable lists.

```
extend module Stdlib;
```

```
module Stack;
class stack[T];
  fun push(s:stack[`T], x:T):void;
  fun pop(s:stack[`T]):T;
  fun top(s:stack[`T]):T;
  fun new_stack[T]():stack[T];

end module Stack;
```

In `queue.diesel`:

Queues are implemented as special interfaces to mutable lists.

```
extend module Stdlib;

module Queue;
class queue[T];
  fun enqueue(t:queue[`T], x:T):void;
  fun dequeue(t:queue[`T]):T;
  fun dequeue(t:queue[`T], if_empty:&():`S):T|S;
  fun new_queue[T]():queue[T];

end module Queue;
```

4.10 Advanced collection

The remainder of this section describes advanced collections and can be skipped upon first reading.

4.10.1 Keyed sets

In `keyed-set.diesel`:

Keyed sets are a space-saving cross between a table and a set. A keyed set is a table where the keys are stored as part of the elements of the table; if the key was already part of the value, then this is more efficient than extracting the key from the value, then storing it separately in a table.

```
extend module Stdlib;

module KeyedSet;
```

Keyed set elements must be subtypes of `keyed_comparable`:

```
abstract class keyed_comparable[Key <= comparable[Key]];
  fun key(:keyed_comparable[`Key]):Key;
  fun has_key(k1:keyed_comparable[`Key], k2:Key):bool;
  fun is_same_key(k1:keyed_comparable[`Key],
                       k2:keyed_comparable[`Key]):bool;
```

Keyed sets also support `removable_` and `extensible_collection` operations such as `remove`, `add`, `add_all`, `add_nonmember`. Keyed sets also support table behavior, plus an associative lookup operation, `match`, which returns the element of the set that has the same key (using `=`) as the second argument (invoking the `if_absent` closure if no such matching element is found).

```
abstract class keyed_set[Key <= comparable[Key],
                          Value <= keyed_comparable[Key]]
                isa table[Key,Value],
                    unordered_collection[Value],
                    -- Value type parameter is covariant:
                    keyed_set[Key, `Value1 >= Value];
```

```
abstract class keyed_set_exactly[Key <= comparable[Key],
                                 Value <= keyed_comparable[Key]]
                isa table_exactly[Key,Value],
                    unordered_collection_exactly[Value],
                    keyed_set[Key, Value];
  fun match(t:keyed_set_exactly[‘Key,‘Value], v:Value,
                   if_absent:&():‘T):Value|T;
  fun match(t:keyed_set_exactly[‘Key,‘Value], v:Value):Value;
```

If values are `comparable` or `hashable`, then so is the collection. Two keyed sets are equal when they include the same elements (regular set equality operations).

```
  extend class keyed_set_exactly[‘Key, ‘Value <= comparable[Value]]
                                isa comparable[keyed_set_exactly[Key,Value]];
  extend class keyed_set_exactly[‘Key, ‘Value <= hashable[Value]]
                                isa hashable[keyed_set_exactly[Key,Value]];
```

As usual, immutable and mutable varieties are defined. Mutable keyed sets also support the adding and removing operations of `extensible_collection`, thereby acting a lot like sets (hence their name), and the storing and removing operations of `m_removable_table`. The store operation for keyed sets is restricted, however, in that the key of the value being stored must match the key where it's being stored, i.e., `store(k, v)` must be identical in effect to `add(v)`.

```
abstract class i_keyed_set[Key <= comparable[Key],
                           Value <= keyed_comparable[Key]]
                isa keyed_set[Key,Value],
                    i_table[Key,Value],
                    i_unordered_collection[Value],
                    -- Value type parameter is covariant:
                    i_keyed_set[Key, ‘Value1 >= Value];
abstract class i_keyed_set_exactly[Key <= comparable[Key],
                                   Value <= keyed_comparable[Key]]
                isa keyed_set_exactly[Key,Value],
                    i_table_exactly[Key,Value],
                    i_unordered_collection_exactly[Value],
                    i_keyed_set[Key, Value];
abstract class m_keyed_set[Key <= comparable[Key],
                           Value <= keyed_comparable[Key]]
                isa keyed_set_exactly[Key,Value],
                    m_removable_table[Key,Value],
                    extensible_collection[Value],
                    m_unordered_collection[Value];
  extend class m_keyed_set[‘Key, ‘Value <= comparable[Value]]
                isa removable_collection[Value];
```

Mutable keyed sets also support a `remove_match` operation that removes the element of the table whose key matches that of the argument element.

```
  fun remove_match(:m_keyed_set[‘Key,‘Value], value:Value,
                         if_absent:&():‘Else):Value|Else;
  fun remove_match(t:m_keyed_set[‘Key,‘Value], value:Value):Value;
```

A linked-list based implementation of `m_keyed_sets`.

```
class list_keyed_set[Key <= comparable[Key],
                     Value <= keyed_comparable[Key]]
```

```
                                                isa m_keyed_set[Key,Value];
  fun new_list_keyed_set[Key <= comparable[Key],
                                Value <= keyed_comparable[Key]](
                                      ):list_keyed_set[Key,Value];

end module KeyedSet;
```

In hash-keyed-set.diesel:

The hashing-based keyed sets require the keys to be hashable, implying that the elements of the keyed set must be subtypes of `keyed_hashable`, not just `keyed_comparable`. `hash_keyed_set` is an implementation of mutable keyed sets using open hashing; `chained_hash_keyed_set` is an implementation of mutable keyed sets based on closed hashing. The hashing-based keyed set implementations allow a guess at a maximum size to be provided when the collection is created. As with all hashing-based implementations, however, the keyed set will automatically resize itself if it grows too large or small.

```
extend module Stdlib;

module HashKeyedSet;
abstract class keyed_hashable[Key <= hashable[Key]]
                                                isa keyed_comparable[Key];
  fun hash_key(k:keyed_hashable[`Key], range:int):int;
class hash_keyed_set[Key <= hashable[Key],
                            Value <= keyed_hashable[Key]]
                    isa m_keyed_set[Key,Value];
fun new_hash_keyed_set[Key <= hashable[Key],
                              Value <= keyed_hashable[Key]](
      ):hash_keyed_set[Key,Value];
fun new_hash_keyed_set[Key <= hashable[Key],
                              Value <= keyed_hashable[Key]](
      size:int):hash_keyed_set[Key,Value];

end module HashKeyedSet;
```

In chained-hash-keyed-set.diesel:

```
extend module Stdlib;

module ChainedHashKeyedSet;
class chained_hash_keyed_set[Key <= hashable[Key],
                                    Value <= keyed_hashable[Key]]
                                            isa m_keyed_set[Key,Value];
  fun new_chained_hash_keyed_set[Key <= hashable[Key],
                                        Value <= keyed_hashable[Key]](
        ):chained_hash_keyed_set[Key,Value];
  fun new_chained_hash_keyed_set[Key <= hashable[Key],
                                        Value <= keyed_hashable[Key]](
        size:int):chained_hash_keyed_set[Key,Value];

end module ChainedHashKeyedSet;
```

In small-keyed-set.diesel:

An implementation of `m_keyed_set` that is space efficient when there are only a few elements, but scales well when there are a large number of elements. Keys of small keyed sets are required to be `hashable`, as in `hash_keyed_sets`.

```
extend module Stdlib;
```

```
module SmallKeyedSet;
class small_keyed_set[K <= hashable[K], V <= keyed_hashable[K]]
                                                        isa m_keyed_set[K,V];
fun shrink_small_set(t:small_keyed_set[‘K,‘V]):void;
fun new_small_keyed_set[K <= hashable[K],
                               V <= keyed_hashable[K]]():small_keyed_set[K,V];
module AbsentKeyedSet;
synonym big_keyed_set[‘K <= hashable[K],‘V <= keyed_hashable[K]] =
  m_keyed_set[K,V]|absent_keyed_set[K,V];
object absent_keyed_set[K <= hashable[K], V <= keyed_hashable[K]];

end module AbsentKeyedSet;

end module SmallKeyedSet;
```

4.10.2 Collectors

In `collector.diesel`:

Sequences support the `||` concatenation operator. However, for long series of concatenations, using `||` many times can be inefficient and imply lots of copying. The `collector` extensible sequence data structure supports accumulating sequences for latter concatenation in one fell swoop.

Collectors can be created (optionally with a non-binding guess as to how many things will be collected together) using new_collector. The infix `&&` operation is the more common way to construct collectors. Collectors are flattened into a vector form using `flat_vector`, and collectors of character sequences can be flattened into a simple string using `flat_string`. To illustrate the use of collectors versus concatenation:

```
    ("hi" && "there" && "bob").flat_string = "hi" || "there" || "bob"
```

```
extend module Stdlib;

module Collector;
class collector[T <= sequence_exactly[S]]
                                        isa ordered_collection_exactly[T];
  fun flat_string(:string|collector[string]):string;
  fun new_collector[T <= sequence_exactly[‘S]]():collector[T];
fun &&(:(‘T <= sequence_exactly[‘S]) | collector[‘T],
              :(‘T <= sequence_exactly[‘S]) | collector[‘T]):collector[T];

end module Collector;
```

4.10.3 Histograms

In `histogram.diesel`:

A `histogram` supports accumulating counts for particular values and then printing out the results in a reasonable fashion. Abstractly, a `histogram` is a mapping from some domain of values to integers. The `increment` operation bumps the count associated with a particular value. To support the histogram's hash-table-based implementation, the values being counted by the `histogram` must be hashable. If the keys also are ordered, then nicer output is possible. The methods `new_unsorted_histogram` vs. `new_histogram` construct the two kinds of histograms.

```
extend module Stdlib;

module Histogram;
class histogram[T <= hashable[T]] isa hash_CR_table[T,integer];
  fun new_unsorted_histogram[T <= hashable[T]]():histogram[T];
  fun new_unsorted_histogram[T <= hashable[T]](t:string):histogram[T];
  fun increment(t:histogram[‘T], x:T):void;
```

```
  fun increment_by_count(t:histogram[`T], x:T, cnt:integer):void;
  fun increment(t:histogram[`T], x:T, elem:any):void;
  fun print_statistics(t:histogram[`T]):string;
  fun print_statistics(t:histogram[`T], some_key:T|string):string;
  fun ordered_keys(t:histogram[`T]):collection[T];
  fun frequency_sorted_print_string(t:histogram[`T]):string;
  fun percent_print_string(t:histogram[`T]):string;
  fun truncated_percent_print_string(t:histogram[`T <= ordered[T]],
                                        over:T):string;
  fun unsorted_distribution[`T <= hashable[T]](
        nm:string, cl:&(increment:&(value:T):void,
                        add_value:&(value:T, elem:any):void):void
        ):histogram[T];
class sorted_histogram[T <= ordered_hashable[T]] isa histogram[T];
  fun new_histogram[T <= ordered_hashable[T]]():histogram[T];
  fun new_histogram[T <= ordered_hashable[T]](t:string):histogram[T];
  fun distribution[`T <= ordered_hashable[T]](
        nm:string,
        cl:&(increment:&(value:T):void,
             add_value:&(value:T, elem:any):void):void
        ):histogram[T];
  fun distribution(nm:string,
                          cl:&(increment:&(value:int):void,
                               add_value:&(value:int, elem:any):void):void
                          ):histogram[int];

end module Histogram;
```

4.10.4 Filtered and mapped views

Views support a transformation of an existing collection data structure, without requiring a copy operation and while preserving mutability. Filtered tables allow viewing an existing table through an arbitrary predicate filter, such as including only a subset of the keys of the table. Mapped tables support viewing an existing table after first processing the keys through a mapping function.

In `filtered.diesel`:

The `view_filtered` methods take a predicate function to filter the keys. The `view_subset` function supports the special case where the filtering predicate is that the key is drawn from a particular set. The mutable filtered table implementation doesn't support adding new bindings through the store function, only updating existing bindings.

```
extend module Stdlib;

module FilteredTable;
class filtered_table[Key,Value,Table <= table_exactly[Key,Value]]
                                              isa table_exactly[Key,Value];
class m_filtered_table[Key,Value,Table <= m_table[Key,Value]]
                isa filtered_table[Key,Value,Table], m_table[Key,Value];
fun view_filtered(t:table_exactly[`Key,`Value], filter:&(Key):bool
                         ):table_exactly[Key,Value];
fun view_subset(t:table_exactly[`Key <= comparable[Key],`Value],
                       keys:set[Key]
                       ):table_exactly[Key,Value];

end module FilteredTable;
```

In `mapped.diesel`:

The `view_mapped` function takes a table mapping `K2` to `V` and a key mapping table mapping `K1` to `K2` and returns a new table that maps from `K1` to `V` transparently. The `view_index_mapped` function takes a table and a mapping from dense integers to the keys of the table (a.k.a. an indexed collection of the keys) and returns a new table mapping from dense integers to the values of the table (a.k.a. an indexed collection of the values). The `view_subrange` functions support similar functionality if the viewed table is an indexed collection and the key map is an interval. The `view_subrange` function is particularly useful for constructing (mutable) views of an existing large indexed collection and then applying standard indexed collection operations to the subrange. For example:

```
    let x:array[string] := ...;
    let rest:m_indexed[string] := x.view_subrange(1, x.length.pred);
    let evens:m_indexed[string] := x.view_subrange(0, x.length.pred, 2);
```

```
extend module Stdlib;

module MappedCollection;
class mapped_table[Key1, Key2, Value] isa table_exactly[Key1, Value];
class m_mapped_table[Key1, Key2, Value]
                isa mapped_table[Key1,Key2,Value], m_table[Key1, Value];
fun view_mapped(t:table[‘Key2, ‘Value],
                      map:table_exactly[‘Key1, Key2]
                      ):table[Key1, Value];
class indexed_table[Key, Value]
    isa mapped_table[int,Key,Value], indexed_exactly[Value];
class m_indexed_table[Key, Value]
    isa indexed_table[Key,Value],
        m_mapped_table[int,Key,Value],
        m_indexed[Value];
fun view_index_mapped(t:table[‘Key, ‘Value],
                            map:indexed_exactly[Key]
                            ):indexed[Value];
fun view_subrange(t:indexed[‘T], from:int):indexed[T];
fun view_subrange(t:indexed[‘T], from:int, to:int):indexed[T];
fun view_subrange(t:indexed[‘T], from:int, to:int, step:int):indexed[T];
```

Views of strings preserve “stringness.”

```
class string_view isa indexed_table[int,char], string;
class m_string_view isa string_view, m_string,
                              m_indexed_table[int,char];
fun view_string_index_mapped(s:string, map:indexed_exactly[int]
                                   ):string;

end module MappedCollection;
```

5 Input/output

5.1 Streams

In `stream.diesel`:

A `stream` is like a collection of values, but the values are accessed in sequence. Streams have an implicit position within the stream of values, pointing between two values (or before the first value or after the last value), and operations on streams are relative to this implicit position.

```
extend module Stdlib;

module Stream;
```

The basic stream data type supports forward reading of a stream of values. The at_end and before_end testers check whether the current position is after the last value in the stream. The forward operation advances the position past the next value in the stream. The next operation reads and returns the next value in the stream, advancing the position as a side-effect; the next_N operation reads and returns the next N elements of the stream. The peek_next operation reads and returns the next element of the stream but does not advance the position; a subsequent next or peek_next operation will return the same value.

```
abstract class stream[T] isa stream[‘S >= T];
fun is_at_end(:stream[‘T]):bool;
fun before_end(s:stream[‘T]):bool;
fun forward(:stream[‘T], at_end:&():void):void;
fun forward(s:stream[‘T]):void;
fun next(:stream[‘T], at_end:&():‘S):T|S;
fun next(s:stream[‘T]):T;
fun next_N(s:stream[‘T], n:int):sequence[T];
fun peek_next(:stream[‘T], at_end:&():‘S):T|S;
fun peek_next(s:stream[‘T]):T;
fun as_collection(s:stream[‘T]):sequence[T];
```

A mutable stream supports changing the value after current position in the stream, and optionally advancing the position past that value. A flush operation makes updates to stream values visible externally, for those implementations of streams such as files that have separate external views.

```
abstract class m_stream[T] isa stream[T];
fun set_peek_next(:m_stream[‘T], :T):void;
fun set_next(s:m_stream[‘T], x:T):void;
fun flush(:m_stream[‘T]):void;
```

A removable_stream supports the remove_next operation, which removes the next element from the stream.

```
abstract class removable_stream[T] isa stream[T],
                                           removable_stream[‘S >= T];
fun remove_next(:removable_stream[‘T]):void;
```

An insertable_stream allows an item (or a collection of items) to be inserted immediately behind the current position of the stream, i.e., the position in the stream is right after the inserted value(s).

```
abstract class insertable_stream[T] isa stream[T];
fun insert(:insertable_stream[‘T], :T):void;
fun insert_all(s:insertable_stream[‘T], c:ordered_collection[T]):void;
```

An extensible_stream allows an item (or a collection of items) to be added to the end of the stream; the position of the stream is moved to the end of the stream after the added items.

```
abstract class extensible_stream[T] isa stream[T];
```

A reversible_stream allows backward motion through the stream. It supports backward-looking functions analogous to the forward-looking operations of the generic stream. Similarly, m_reversible_stream supports backwards-looking mutation operations.

```
abstract class reversible_stream[T] isa stream[T],
                                            reversible_stream[‘S >= T];
fun is_at_start(:reversible_stream[‘T]):bool;
fun after_start(s:reversible_stream[‘T]):bool;
fun backward(:reversible_stream[‘T], at_start:&():void):void;
```

```
fun backward(s:reversible_stream[‘T]):void;
fun prev(:reversible_stream[‘T], at_start:&():‘S):T|S;
fun prev(s:reversible_stream[‘T]):T;
fun peek_prev(:reversible_stream[‘T], at_start:&():‘S):T|S;
fun peek_prev(s:reversible_stream[‘T]):T;
abstract class m_reversible_stream[T] isa reversible_stream[T],
                                                m_stream[T];
fun set_peek_prev(:m_reversible_stream[‘T], :T):void;
fun set_prev(s:m_reversible_stream[‘T], x:T):void;
```

A `positionable_stream` supports querying and setting the position in the stream explicitly. The length of the stream can be determined also.

```
abstract class positionable_stream[T] isa reversible_stream[T],
                                                positionable_stream[‘S >= T];
fun position(:positionable_stream[‘T]):int;
fun set_position(:positionable_stream[‘T], :int, off_end:&():void):void;
fun set_position(s:positionable_stream[‘T], p:int):void;
fun to_start(s:positionable_stream[‘T]):void;
fun to_end(s:positionable_stream[‘T]):void;
fun view_indexed(:positionable_stream[‘T]):indexed[T];
abstract class m_positionable_stream[T] isa positionable_stream[T],
                                                  m_reversible_stream[T];

end module Stream;
```

5.2 stream_views ; view_stream

In `indexed-stream.diesel`:

Stream views of indexed collections are fully positionable, but not insertable, extensible, or removable. (Stream views of arrays could be made extensible.)

```
extend module Stdlib;

module IndexedStreamView;
class indexed_stream_view[T] isa positionable_stream[T];
class m_indexed_stream_view[T] isa m_positionable_stream[T],
                                       indexed_stream_view[T];

end module IndexedStreamView;
```

In `list-stream.diesel`:

Stream views of lists are not positionable or extensible but are insertable and removable.

```
extend module Stdlib;

module ListStreamView;
class list_stream_view[T] isa m_stream[T],
                                 removable_stream[T], insertable_stream[T];
class m_list_stream_view[T] isa list_stream_view[T];

end module ListStreamView;
```

In `string-stream.diesel`:

```
extend module Stdlib;

module StringStream;
  fun new_string_stream(s:string):positionable_stream[char];

end module StringStream;
```

5.3 Random numbers

Random numbers are supported through the `random_stream` interface.

In `randstream.diesel`:

A `random_stream` is an infinitely-long stream of ints in the range 0 to range-1 inclusive. The initial seed of the random stream can be set upon creation, to generate reproducible random streams. (But multiple random streams cannot each have their own seed set independently! There's a single global seed shared by all random streams.)

```
extend module Stdlib;

module RandomStream;
class random_stream isa stream[int];
fun get_rand():int;
fun set_rand_seed(i:int):void;
fun new_rand_stream(i:int):random_stream;
fun new_rand_stream(i:int, j:int):random_stream;
fun random_vector(len:int, range:int):vector[int];

end module RandomStream;
```

5.4 Unix files

In `file.diesel`:

A `file` object acts like a mutable, extensible, positionable stream of characters, as well as supporting lots of standard file I/O operations.

Files can be opened, given the name of the file and an open mode, using `open_file`. The optional `if_error` closure taken by `open_file` and many other file operations is invoked if there was a standard OS error during the operation, passing a string describing the kind of error. The `os_error` function invokes `error` with an appropriate error message derived from the error string and a user-supplied message. The `nonfatal_os_error` also prints the error message, but then successfully returns to the caller.

The three standard files can be returned by the `stdin`, `stdout`, and `stderr` functions. Six functions are available to query properties of the file: whether or not the file is readable and/or writable and whether or not writes always append. The `read_partial` functions read up to `size` characters into a buffer, starting at index `from`; they return how many characters actually were read. The `read` functions repeatedly call `read_partial` until all requested characters are read. The `read_line` functions work like `read`, except that they stop reading after they've seen (and copied to the buffer) a newline character. The `read_whole_text_file` functions create and return a string containing the whole file's contents; it correctly handles text files with varying CR/LF conventions on different platforms. The `write` functions write their argument character buffer (optionally copying only `size` characters) to the file. Collectors can be written to a file directly, more efficiently than first flattening the collector into a string. Individual characters can also be written to a file.

The `mod_time` operations return the modification timestamp of the file; `get_mod_time` is a convenience in case the file hasn't been opened yet. (The `time` data structure supports parsing the timestamp integer.)

In addition to the other stream-style operations, `files` support testing whether they have detected the end of the file (subtly different than actually being at the end of the file) and performing `lseek`-style repositioning relative either to the start of the file, the current position, or the end of the file. Note that file positions and file lengths are in terms of raw characters in the underlying file, ignoring any CR/LF conversion that might be performed by `read` operations.

```
extend module Stdlib;

module File;
fun stdin():file;
fun stdout():file;
```

```
fun stderr():file;
abstract class open_mode;
object open_for_reading isa open_mode;
object create_for_writing isa open_mode;
object open_for_append isa open_mode;
object open_for_update isa open_mode;
object create_for_update isa open_mode;
object open_for_appending_update isa open_mode;
object open_binary_for_reading isa open_mode;
object create_binary_for_writing isa open_mode;
object open_binary_for_append isa open_mode;
object open_binary_for_update isa open_mode;
object create_binary_for_update isa open_mode;
object open_binary_for_appending_update isa open_mode;
fun open_file(n:string, m:open_mode):file;
fun open_file(n:string, m:open_mode, if_error:&(string):file):file;
fun file_name(f:file):string;
fun is_readable(f:file):bool;
fun is_unreadable(f:file):bool;
fun is_writable(f:file):bool;
fun is_unwritable(f:file):bool;
fun is_read_write(f:file):bool;
fun is_append(f:file):bool;
fun read(f:file, buffer:m_indexed[char], size:int):void;
fun read(f:file, buffer:m_indexed[char], size:int,
                if_error:&(string):void):void;
fun read_whole_text_file(f:file):string;
fun read_whole_text_file(f:file, if_error:&(string):string):string;
fun read_partial(f:file, buffer:m_indexed[char], from:int, size:int
                        ):int;
fun read_partial(f:file, buffer:m_indexed[char], from:int, size:int,
                        if_error:&(string):int):int;
fun read_line(f:file):string;  -- returns the line's contents with no trailing
n; "" if the line is empty
fun read_line(f:file, if_eof:&():string):string;
fun read_line(f:file, if_eof:&():string, if_error:&(string):string
                     ):string;
fun read_line_into(f:file, buffer:m_indexed[char], size:int):int;
fun read_line_into(f:file, buffer:m_indexed[char], size:int,
                          if_error:&(string):int):int;
fun write_char(f:file, x:char):void;
fun write_line(f:file):void;
fun write_line(f:file, buffer:indexed[char]):void;
fun write(f:file, buffer:indexed[char]):void;
fun write(f:file, buffer:indexed[char], size:int):void;
fun write(f:file, buffer:indexed[char], size:int,
                 if_error:&(string):void):void;
fun print(s:indexed[char], f:file):void;
fun write_to_file(o:any, fname:string):void;
fun write_to_file(o:any, fname:string, if_error:&(string):none):void;
fun write_collector(f:file, c:collector[`T <= indexed[char]]):void;
fun write_collector(f:file, c:collector[`T <= indexed[char]],
                           if_error:&(string):void):void;
fun detected_eof(f:file):bool;
fun close(f:file):void;
fun close(f:file, if_error:&(string):void):void;
```

```
fun get_mod_time(f_name:string):integer;
fun get_mod_time(f_name:string, if_error:&(string):integer):integer;
fun mod_time(f:file):integer;
fun mod_time(f:file, if_error:&(string):integer):integer;
fun mod_time_internal(f:file, if_error:&(string):int):int;
fun error_string(i:int):string;
fun os_error(err:string, s:string):none;
fun nonfatal_os_error(err:string, s:string):void;
fun position(f:file, if_error:&(string):int):int;
abstract class position_mode;
object from_start isa position_mode;
object from_current_position isa position_mode;
object from_end isa position_mode;
fun set_file_position(f:file, offset:int, if_error:&(string):void):void;
fun set_position_relative(f:file, offset:int, from:position_mode):void;
fun set_position_relative(f:file, offset:int, from:position_mode,
                                 if_error:&(string):void):void;
fun flush(f:file, if_error:&(string):void):void;
fun file_exists(s:string):bool;   -- return whether the named file exists
fun path_name(t:string):string;
fun dirname(t:string):string;
fun strip_leading_path(t:string):string;
fun basename(t:string):string;
fun canonicalize_filename(s:string):string;
fun canonicalize_dirname(s:string):string;
fun shrink_filename(s:string):string;   -- use ~ in place of the current user's home directory
fun is_abs_filename(s:string):bool;
```

The `find_file` operation takes a file name and a directory search path and returns the full path name for the first file that exists under that name in the search path.

```
fun find_file(base_name:string, dirs:ordered_collection[string],
                     if_fail:&(string):string):string;
fun find_file(base_name:string, dirs:sequence[string]):string;
fun expand_filename(s:string):string;   -- replace leading ~.. with its expansion
fun expand_filename(s:string, if_error:&(string):string):string;
fun write_object_to_file_name(obj:any, f_name:string):void;
fun write_object_to_file_name(obj:any, f_name:string,
                                     if_error:&(string):void):void;
fun write_object_to_file(x:any, f:file, :bool | &(string):void):void;
fun write_object_to_file(x:any, f:file, use_bs:bool,
                                if_error:&(string):void):void;
fun read_object_from_file_name(f_name:string):dynamic;
fun read_object_from_file_name(f_name:string, use_bs:bool):dynamic;
fun read_object_from_file_name(f_name:string, use_bs:bool,
                                      if_error:&(string):dynamic):dynamic;
fun read_object_from_file(f:file, :bool | &(string):dynamic):dynamic;
fun read_object_from_file(f:file, use_bs:bool,
                                 if_error:&(string):dynamic):dynamic;

end module File;
```

6 Miscellaneous

6.1 ask

In `ask.diesel`:

```
extend module Stdlib;

module Ask;
```

The `ask` method prints out a prompt on `stdout` and returns the result typed in on `stdin`. The `ask_yes_no` method returns true iff the first character of the response begins with Y or y.

```
fun ask(prompt:string):string;
fun ask_yes_no(prompt:string):bool;

end module Ask;

module BlockInput;
fun get_block_input():string;

end module BlockInput;

module Split;
fun split_input(s:string, split_ch:char,
                       needs_continuation:&(extensible_sequence[string]
                                            ):extensible_sequence[string]
                       ):extensible_sequence[string];
fun split_input(s:string, split_ch:char):extensible_sequence[string];

end module Split;
```

6.2 Time and date

In `time.diesel`:

The `date_info` object provides hygienic access to the `current_time` function. A new `date_info` representing the current time is returned by the `date` function; a `date_info` object for a given time is constructed by the `new_date_info` method. Many aspects of the current time and date can be queried; the `_shortname` versions return 3-letter abbreviations of their `_name` equivalents. The numbers returned are zero-based (i.e., January is month 0 and Sunday is day 0). The `print_string` output for a `date_info` object is identical to the output of the Unix date command.

```
extend module Stdlib;

module TimeAndDate;
let earliest_time:integer;
let latest_time:integer;
fun int_time_to_integer_time(t:int):integer;
fun integer_time_to_int_time(t:integer):int;
fun integer_time_to_int_time(t:integer, if_error:&(string):int):int;
fun current_time():integer;
fun real_time(cl:&():void):int;
class date_info isa comparable[date_info];
  fun new_date_info(time:integer):date_info;
  fun date():date_info;
  fun seconds(d:date_info):int;
  fun minutes(d:date_info):int;
  fun hours(d:date_info):int;
  fun day_of_month(d:date_info):int;
  fun month_of_year(d:date_info):int;
  fun month_of_year_name(d:date_info):string;
  fun month_of_year_shortname(d:date_info):string;
  fun year(d:date_info):int;
```

```
  fun day_of_week(d:date_info):int;
  fun day_of_week_name(d:date_info):string;
  fun day_of_week_shortname(d:date_info):string;
  fun day_of_year(d:date_info):int;
  fun is_daylight_savings_time(d:date_info):bool;
  fun time_zone_name(d:date_info):string;

end module TimeAndDate;
```

6.3 text_lines

In `text-lines.diesel`:

The `text_lines` data structure represents a series of lines with newlines separating them. The `new_text_lines` function breaks a string after newlines into separate text lines. The `lines` function provides access to the lines of text. The `indent` function adds `count` spaces to the front of each text line. The `as_collector` and `as_string` functions convert the `text_lines` object to a `collector` or a flat `string` representation, with embedded newlines.

```
extend module Stdlib;

module TextLines;
class text_lines;
  field lines(:text_lines):array[string];
  fun new_text_lines(s:string):text_lines;
  fun indent(tl:text_lines, i:int):void;
  fun as_collector(tl:text_lines):collector[string];

end module TextLines;
```

6.4 2-d matrices

In `matrix.diesel`:

A `matrix` is a two-dimensional indexable collection.

Matrices support querying the number of rows and columns of a matrix, extracting an element, a row, or a column of the matrix. The `do` and `indices_do` methods support iterating over the matrix. Conformable matrices can be added and multiplied. A matrix of comparable values is itself comparable, pointwise. Mutable matrices support changing a given matrix element.

One concrete implementation of matrices exists, based on representing matrices by a vector of vectors. The `new_vector_matrix_init` functions supports creating a new `vector_matrix` of a given size with its elements initialized as specified by the `init_fn` closure.

```
extend module Stdlib;

module Matrix;
abstract class matrix[T];
  fun num_rows(:matrix[`T]):int;
  fun num_cols(:matrix[`T]):int;
  fun fetch(:matrix[`T], row:int, col:int):T;
  fun row(m:matrix[`T], row:int):indexed[T];
  fun col(m:matrix[`T], col:int):indexed[T];
  fun rows(m:matrix[`T]):indexed[indexed[T]];
  fun indices_do(m:matrix[`T], c:&(int,int):void):void;
  fun do(m:matrix[`T], c:&(int,int,T):void):void;
  fun +(m1:matrix[`T <= num], m2:matrix[T]):matrix[T];
  fun *(m1:matrix[`T >= int <= num], m2:matrix[T]):matrix[T];
```

```
  fun *_ugly(m1:matrix[‘T >= int <= num], m2:matrix[T]):matrix[T];
  extend class matrix[‘T <= comparable[T]] isa comparable[matrix[T]];
  fun copy_init(m:matrix[‘T], num_rows:int, num_cols:int,
                      init:&(int,int):T):matrix[T];
  fun copy_init[T](:matrix[T], num_rows:int, num_cols:int,
                         init:&(int,int):T):matrix[T];
  fun copy_mutable_init(m:matrix[‘T], num_rows:int, num_cols:int,
                              init:&(int,int):T):m_matrix[T];
  fun copy_mutable_init[T](:matrix[T], num_rows:int, num_cols:int,
                                 init:&(int,int):T):m_matrix[T];
abstract class m_matrix[T] isa matrix[T];
  fun store(:matrix[‘T], row:int, col:int, value:T):void;
class vector_matrix[T] isa m_matrix[T];
  fun new_vector_matrix_init[T](num_rows:int, num_cols:int,
                                       init:&(int,int):T):m_matrix[T];

end module Matrix;
```

6.5 Graphs and partial orders

In `graph.diesel`:

Basic graph ADT. A graph is a set of nodes, and each node has a set of in edges and a set of out edges. Clients can just provide concrete subclasses of graph_node, implementing = and hash, or they can subclass graph_edge and/or graph also.

```
extend module Stdlib;

module Graph;
abstract class graph_node[Node <= graph_node[Node,Edge],
                                 Edge <= graph_edge[Node,Edge]]
                                        isa hashable[graph_node[Node,Edge]];
  field in_edges(:graph_node[‘Node,‘Edge]):m_set[Edge];
  field out_edges(:graph_node[‘Node,‘Edge]):m_set[Edge];
abstract class graph_node[Node <= graph_node[Node]]
                                isa graph_node[Node, graph_edge[Node]],
                                    hashable[graph_node[Node]];
abstract class graph_edge[Node <= graph_node[Node,Edge],
                                 Edge <= graph_edge[Node,Edge]]
                                        isa comparable[graph_edge[Node,Edge]];
  var field from_node(:graph_edge[‘Node,‘Edge]):Node;
  var field to_node(:graph_edge[‘Node,‘Edge]):Node;
  fun add_edge(e:‘Edge <= graph_edge[‘Node,‘Edge]):void;
  fun remove_edge(e:‘Edge <= graph_edge[‘Node,‘Edge]):void;
class graph_edge[Node <= graph_node[Node]]
                                isa graph_edge[Node, graph_edge[Node]],
                                    comparable[graph_edge[Node]];
  fun new_graph_edge(f:‘Node <= graph_node[Node], t:Node):graph_edge[Node];
class graph[Node <= graph_node[Node,Edge],
                   Edge <= graph_edge[Node,Edge]];
  fun add_node(g:graph[‘Node,‘Edge], node:Node):void;
  fun remove_node(g:graph[‘Node,‘Edge], node:Node):void;
  fun add_edge(g:graph[‘Node,‘Edge], e:Edge):void;
  fun remove_edge(g:graph[‘Node,‘Edge], e:Edge):void;
  fun print_header(g:graph[‘Node,‘Edge]):string;
  fun print_headers(g:graph[‘Node,‘Edge]):string;
  fun new_graph[Node <= graph_node[Node,‘Edge]]():graph[Node,Edge];
```

```
end module Graph;
```

In `partial-order.diesel`:

Partial order graph type. Supports downwards and upwards traversals. Clients can just provide concrete subclasses of partial_order_node, implementing =, hash, and ¡.

```
extend module Stdlib;

module PartialOrder;
abstract class partial_order_node[Node <= partial_order_node[Node,Edge],
                                  Edge <= partial_order_edge[Node,Edge]]
                isa graph_node[Node, Edge],
                    hashable[partial_order_node[Node,Edge]],
                    partially_ordered[partial_order_node[Node,Edge]];
  fun up_edges(t:partial_order_node[`Node,`Edge]):set[Edge];
  fun down_edges(t:partial_order_node[`Node,`Edge]):set[Edge];
  fun is_top(t:partial_order_node[`Node,`Edge]):bool;
  fun is_bottom(t:partial_order_node[`Node,`Edge]):bool;
  fun up_nodes_do(t:partial_order_node[`Node,`Edge],
                         bl:&(Node):void):void;
  fun down_nodes_do(t:partial_order_node[`Node,`Edge],
                           bl:&(Node):void):void;
  fun order_print_string(t:partial_order_node[`Node,`Edge]):string;
abstract class partial_order_node[Node <= partial_order_node[Node]]
                isa partial_order_node[Node, partial_order_edge[Node]],
                    hashable[partial_order_node[Node]],
                    partially_ordered[partial_order_node[Node]];
abstract class partial_order_edge[Node <= partial_order_node[Node,Edge],
                                  Edge <= partial_order_edge[Node,Edge]]
                                  isa graph_edge[Node,Edge];
  fun down_node(e:partial_order_edge[`Node,`Edge]):Node;
  fun up_node(e:partial_order_edge[`Node,`Edge]):Node;
class partial_order_edge[Node <= partial_order_node[Node]]
                isa partial_order_edge[Node, partial_order_edge[Node]];
  fun new_partial_order_edge(down:`Node <= partial_order_node[Node],
                                    up:Node):partial_order_edge[Node];
abstract class partial_order[Node <= partial_order_node[Node,Edge],
                                    Edge <= partial_order_edge[Node,Edge]]
                                                isa graph[Node, Edge];
  field tops(:partial_order[`Node,`Edge]):m_set[Node];
  field bottoms(:partial_order[`Node,`Edge]):m_set[Node];
  fun add_partial_order_edges(t:partial_order[`Node,`Edge]):void;
  fun top_down_do(t:partial_order[`Node,`Edge], cl:&(Node):void):void;
  fun bottom_up_do(t:partial_order[`Node,`Edge], cl:&(Node):void):void;
  fun ordered_strictly_below(t:partial_order[`Node,`Edge],
                                    n1:Node, n2:Node):bool;
  fun ordered_below(t:partial_order[`Node,`Edge],
                           n1:Node, n2:Node):bool;
  fun ordered_strictly_above(t:partial_order[`Node,`Edge],
                                    n1:Node, n2:Node):bool;
  fun ordered_above(t:partial_order[`Node,`Edge],
                           n1:Node, n2:Node):bool;
  fun clear_marks(g:partial_order[`Node,`Edge]):void;
  fun new_partial_order_edge(:partial_order[`Node,`Edge],
                                    down:Node, up:Node):Edge;
class partial_order[Node <= partial_order_node[Node]]
                isa partial_order[Node, partial_order_edge[Node]];
```

```
  fun new_partial_order[Node <= partial_order_node[Node]](
        ):partial_order[Node];

end module PartialOrder;
```

6.6 System operations

In `system.diesel`:

```
extend module Stdlib;

module System;
fun exit(error_code:int):none;
fun object_size(obj:any):int;
fun individual_object_size(obj:any):indexed[int];
fun object_size_histogram(obj:any):int;
fun PIC_statistics():void;
fun detailed_PIC_statistics():void;
fun cpu_time():int;  -- Current CPU time in milliseconds.
fun time(closure:&():void):int;  -- CPU execution time of closure in milliseconds
fun time_value(closure:&():`T):pair[int,T];  -- return both the time and the closure's result
fun benchmark_closure(cls:&():void):void;
```

The `system` function invokes the Unix `system` system call with the given command, and passes the returned value to the user. The `if_error` closure is invoked if the system call returns a non-zero result.

```
fun system(s:string):int;
fun system(s:string, if_error:&(i:int):int):int;
fun breakpoint():void;
fun sys_breakpoint():void;
object argv isa i_indexed_exactly[string];  -- Unix command-line arguments
```

Unix environment variables can be read and modified.

```
object env isa m_table_like[string,string];
fun fetch_internal(t:env, name:vstring):string;
fun store_internal(t:env, n:vstring, v:vstring):void;
fun garbage_collect():void;
fun print_heap():void;
fun process_size():int;
fun compile_date():string;
fun zero_runtime_counters():void;
fun print_and_zero_runtime_counters():void;
fun profiling_on():void;
fun profiling_off():void;
fun profile(c:&():`T):T;
fun profile(b:bool, c:&():`T):T;

end module System;
```

6.7 Reflection

In `msg.diesel`:

This file includes primitives allowing the Diesel program access to the run-time system's compiled code and method lookup tables, thus supporting reflection.

```
extend module Stdlib;
```

```
module Reflection;
module Message;
fun send(msg_name:string, num_params:int,
                args:ordered_collection[dynamic]):dynamic;
fun send(msg_name:string, num_params:int,
                args:ordered_collection[dynamic],
                if_error:&():dynamic):dynamic;
fun field_init_send(msg_name:string, num_params:int,
                           args:ordered_collection[dynamic],
                           if_error:&():dynamic):dynamic;
fun prim_resend(msg_name:string, num_params:int,
                       args:ordered_collection[dynamic],
                       dirs:ordered_collection[dynamic],
                       is_undirected:bool, if_error:&():dynamic):dynamic;
fun directed_field_init_send(msg_name:string, num_params:int,
                                    args:ordered_collection[dynamic],
                                    dirs:ordered_collection[dynamic],
                                    if_error:&():dynamic):dynamic;

end module Message;

module Inheritance;
fun type_id(t:any):int;
fun inherits_from(obj1:any, obj2:any):bool;

end module Inheritance;

module Breakpoint;
fun set_breakpoint(msg_name:string):void;
fun show_breakpoints():void;

end module Breakpoint;

end module Reflection;
```

In `env.diesel`:

`evaluation_env`s support Diesel program access to its runtime state, for debugging and fast expression evaluation purposes.

```
extend module Stdlib;

extend module Reflection;

module Environment;
abstract class evaluation_env;
  fun lexically_enclosing_env(:evaluation_env,
                                     if_none:&():evaluation_env
                                     ):evaluation_env;
  fun lookup(e:evaluation_env, s:string, num_params:int,
                    if_absent:&():dynamic,
                    if_error:&(string):dynamic):dynamic;
  fun lookup_assign(e:evaluation_env, s:string, num_params:int,
                           value:dynamic,
                           if_absent:&():void, if_error:&(string):void):void;
  fun find_defining_env(e:evaluation_env, s:string, num_params:int,
                               if_absent:&():evaluation_env,
                               if_error:&(string):evaluation_env
```

```
                              ):evaluation_env;
  fun fetch(:evaluation_env, s:string, num_params:int,
                 if_absent:&():dynamic, if_error:&(string):dynamic):dynamic;
  fun fetch_object(:evaluation_env, s:string, num_params:int,
                        if_absent:&():dynamic, if_error:&(string):dynamic
                        ):dynamic;
  fun assign(:evaluation_env, s:string, num_params:int, value:dynamic,
                  if_absent:&():void, if_error:&(string):void):void;
  fun defines_var(:evaluation_env, s:string, num_params:int,
                       if_error:&(string):bool):bool;
  fun add_var_decl(e:evaluation_env, name:string, is_constant:bool,
                        type_annotation:string, value:dynamic,
                        if_error:&(string):none):void;
  fun add_var_decl(e:evaluation_env, name:string, num_params:int,
                        is_constant:bool, is_abstract:bool,
                        type_annotation:string, value:dynamic,
                        if_error:&(string):none):void;
  fun local_vars_do(:evaluation_env,
                         cl:&(name:string, is_constant:bool,
                              type_annotation:string):void):void;
  fun decl_context_string(:evaluation_env):string;
  fun is_static_env(:evaluation_env):bool;
  fun module_name(e:evaluation_env):string;
  fun module_name(:evaluation_env, if_error:&():string):string;
object empty_env isa evaluation_env;
abstract class debuggable_env isa evaluation_env;
  fun debugger(env:debuggable_env, print_frame:bool):void;
object global_env isa debuggable_env;
  var field extensions(e:global_env):evaluation_env;
fun create_anon_object(r:global_env, parent:dynamic,
                            if_error:&(string):dynamic):dynamic;
fun create_named_object(r:global_env, s:string, num_params:int,
                             is_abstract:bool,
                             parents:indexed[dynamic],
                             if_present:&():dynamic,
                             if_error:&(string):dynamic):dynamic;
abstract class runtime_env isa debuggable_env;
fun current_env():runtime_env;
fun my_caller(if_none:&():`T):runtime_env|T;
fun my_caller():runtime_env|global_env;
fun caller(r:runtime_env, if_none:&():`T):runtime_env|T;
fun extend_method_table(meth_name_oop:string, num_params:int,
                             specializers:ordered_collection[dynamic],
                             method_object:runtime_extension_method,
                             interpret_interrupts:bool,
                             if_error:&():bool):bool;

end module Environment;

end module Reflection;

end module Stdlib;
```

6.8 Application hooks

In app.diesel:

```
extend module Stdlib;
```

```
module App;
```

The `generic_app` object is a place for libraries to put default behavior that can be customized by individual applications.

```
object generic_app;
```

The `app` variable is what libraries should refer to to get the current application object. Individual applications (and optional libraries too) should define children of `generic_app` that override the behavior of `generic_app`, and update the `app` variable to hold that object. Since the `app` variable's type is just `generic_app`, only generic operations are allowed on `app`. For client-specific operations, the client-specific objects should be referred to directly (or the client-specific operations should have appropriate default behavior introduced on `generic_app`, and then overridden by the client-specific operations).

```
let var app:generic_app;
```

For programs that have multiple "applications", e.g. the compiler which has two different versions of the typechecker "application", the `using_app` control structure can be used to temporarily switch to a particular application object during evaluation of a block of code.

```
fun using_app(new_app:generic_app, cl:&():`T):T;
```

```
end module App;
```

6.9 Utilities

In `utilities.diesel`:

```
extend module Stdlib;
```

```
module Utilities;
fun check_for_breakpoint(:generic_app):void;
shared var field debug_on_error(:generic_app):bool;
fun warning_msg(t:generic_app, msg:string):void;
fun warning_msg(msg:ordered_collection[any]):void;
fun print_msg(msg:ordered_collection[any]):void;
fun print_msg_no_CR(msg:ordered_collection[any]):void;
fun overwrite_msg(s:string):void;
fun running_under_emacs(:generic_app):bool;
fun cond_msg(op:int, level:int, cl:&():ordered_collection[any]):void;
fun cond_msg_no_CR(op:int, level:int, cl:&():ordered_collection[any]
                          ):void;
fun show_counter(counter:int):void;
fun step_show_counter(counter:int, step:int):void;
fun finish_show_counter():void;
```

7 Precedence of binary operators

Figure 2 identifies the precedence groups for the binary operators defined in the standard library and indicates which precedence groups take precedence over lower precedence groups (groups higher up have higher precedence than groups lower down, which they point to). All precedence groups have left associativity except for the central group of comparison operators which are non-associative and the `**` exponentiation operator which is right-associative.

The `+_ov` et al. operators have the same precedence as the corresponding non-`_ov` operators.

Explicit parenthesization is required in expressions which mix operators that are either non-associative or unordered by the precedence partial order.

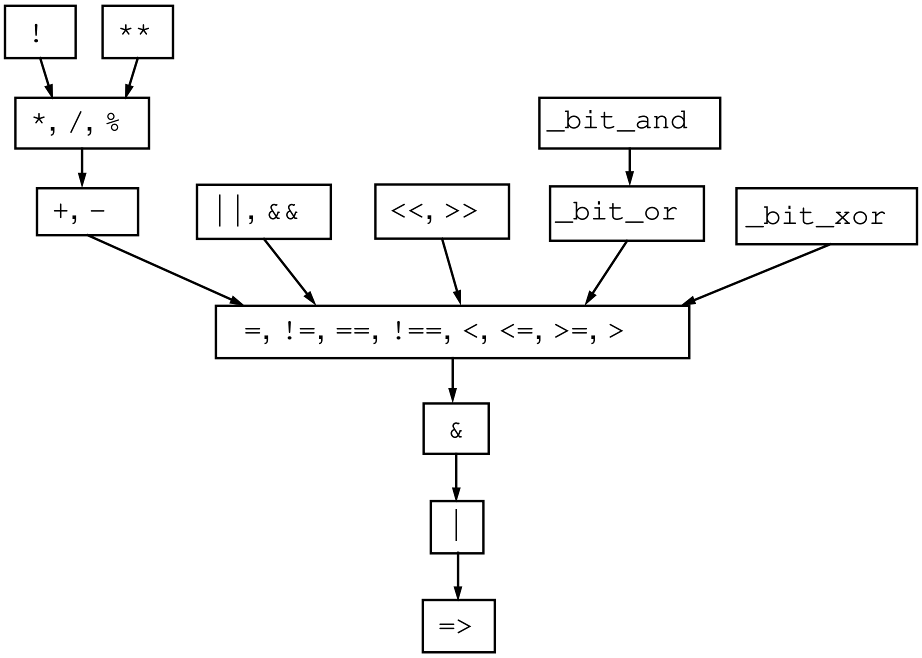

Figure 2: Precedence groups for binary operators; higher groups have higher precedence.

Index