Cecil 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 3.0
December 9, 1998

Contents

1 Introduction

The Cecil standard library defines a collection of data structures and control structures used by most Cecil 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 which make up the noeval standard library. They are available to a program which explicitly includes the file `prelude.noeval.cecil` or which is compiled by the Vortex compiler with the `stdlib` (default) or `noevalstdlib` standard library levels. The "How To Use Vortex" document describes the additional functionality of the evaluator which is available to each program compiled at the stdlib level. The small standard library (`small`) is a part of the `noeval` library, but its independent use is discouraged.

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 and triple

- 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 Cecil code declaring the objects and methods. Method bodies are not shown; the headers show "types" of arguments and the result. 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. In most cases, a representation and its most specific type are declared in a single `object` declaration, inheritance and subtyping in an `isa` declaration. For such objects and inheritance/subtyping, the words "class" and "subclass" will be used occasionally. Template objects are sometimes called "concrete classes" since they may be instantiated at runtime.

1.1 Library code conventions

This section may be skipped on first reading.

Some notation used in the standard library differs from that 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 Cecil language to indicate parameterization of objects and types. For example, printing out an array element (or, more generally, a table element) can be done like this: `(a!i).print;`.

Creation of new anonymous instances of objects at runtime (which corresponds to using the `new` constructor for classes in languages such as C++), is typically achieved by `new_` methods which encapsulate the object constructor expressions. The name of such a method usually starts with `new_` followed by the name of the (template) object whose instance is being created; sometimes a suffix is used to distinguish between different versions. However, this is merely a convention. (The only restriction imposed by the language design is that no anonymous instances of abstract objects may be created at runtime, since these objects are used to define the interfaces and share common code between their subclasses.) Notable exceptions from this convention are methods that construct `cons` cells, `pair`s, `triple`s, and `quadruple`s, which don't start with `new_` for convenience. Here are some examples:

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

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

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

Many operations which 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 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
method fetch(t:table[‘Key,‘Value], key:Key):Value;

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

2 Basic data types and operations

2.1 Maximal and minimal types, identity testing, and printing

```
concrete object void;
```

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

```
abstract object any;
```

`any` is the supertype of all other non-`void` types, implicitly.

```
type none;
```

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 the subtype of all other types; there is no object of this type.

```
type dynamic;
```

The `dynamic` type disables static type checking. It's used to explicitly declare the type of a variable that cannot be statically checked. An omitted type declaration defaults to `dynamic`.

In `general.cecil`:

```
method ==(l:any, r:any):bool;  -- object identity test; very low-level, but fast
method !==(l:any, r:any):bool;  -- not ==
```

All objects can be compared for identity, using a low-level implementation-dependent test. 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_`.

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

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 `unix_files`.)

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

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

```
concrete object not_defined;
```

The `not_defined` object can be used as a dummy object if there isn't a real one to use, e.g., if you have to have an uninitialized variable or field or want to reflect whether or not some value is present. `not_defined` is therefore used in places where a NULL pointer might be used in other languages. In general, it is poor style to use `not_defined`. It is better to define an application-specific "absent" object, integrated into a little class hierarchy of present or absent data, with appropriate application- specific behavior attached to the "absent" object.

2.2 Equality, ordering, and hashing

In `comparable.cecil`:

```
abstract object comparable[T <= comparable[T]];
  extend type comparable[‘T] subtypes comparable[‘S <= T];
signature =(x1:‘T <= comparable[T], x2:‘T <= comparable[T]):bool;  -- equal (abstract) values?
method !=(x1@:‘T <= comparable[T], x2@:‘T <= comparable[T]):bool;  -- not =
```

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 a low-level representation equality 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_`.

```
abstract object partially_ordered[T <= partially_ordered[T]]
      isa comparable[T];
  extend type partially_ordered[‘T] subtypes partially_ordered[‘S <= T];
signature <(x1:‘T <= partially_ordered[T],
            x2:‘T <= partially_ordered[T]):bool;  -- x1 ordered below x2?
method <=(x1@:‘T <= partially_ordered[T],
          x2@:‘T <= partially_ordered[T]):bool;  -- < or =
method > (x1@:‘T <= partially_ordered[T],
          x2@:‘T <= partially_ordered[T]):bool;  -- x2 < x1?
method >=(x1@:‘T <= partially_ordered[T],
          x2@:‘T <= partially_ordered[T]):bool;  -- > or =
```

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`.

```
abstract object ordered[T <= ordered[T]] isa partially_ordered[T];
  extend type ordered[‘T] subtypes ordered[‘S <= T];
method <=(x1@:‘T <= ordered[T], x2@:‘T <= ordered[T]):bool;
```

```
method >=(x1@:`T <= ordered[T], x2@:`T <= ordered[T]):bool;
method min (x1:`T <= ordered[T], x2:`T <= ordered[T]):T;
method max (x1:`T <= ordered[T], x2:`T <= ordered[T]):T;
method compare(x1@:`T <= ordered[T], x2@:`T <= ordered[T],
               if_less:&():`S, if_equal:&():`S, if_greater:&():`S
               ):S;
abstract object ordered_using_compare[T <= ordered[T]] isa ordered[T];
 extend type ordered_using_compare[`T] subtypes ordered_using_compare[`S<=T];
method =(x1@:`T <= ordered_using_compare[T],
         x2@:`T <= ordered_using_compare[T]):bool;
method <(x1@:`T <= ordered_using_compare[T],
         x2@:`T <= ordered_using_compare[T]):bool;
method compare(x1@:`T <= ordered_using_compare[T],
               x2@:`T <= ordered_using_compare[T],
               if_less:&():`S, if_equal:&():`S, if_greater:&():`S
               ):S;
```

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.

```
abstract object hashable[T <= hashable[T]] isa comparable[T];
  extend type hashable[`T] subtypes hashable[`S <= T];
signature hash(hashable[`T], range:int):int;
```

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)`.

```
abstract object ordered_hashable[T <= ordered_hashable[T]] isa ordered[T],
                                                              hashable[T];
  extend type ordered_hashable[`T] subtypes ordered_hashable[`S <= T];
```

An ordered_hashable object is both hashable and ordered.

2.3 Numbers

In `number.cecil`:

```
abstract object num isa ordered_hashable[num];
signature as_int(num):integer;
method as_float(n@:num):float;
signature as_single_float(num):single_float;
signature as_double_float(num):double_float;
method =(l@:num,r@:num):bool;
method <(l@:num,r@:num):bool;
signature +(`T <= num, T):T;  -- standard arithmetic
signature -(`T <= num, T):T;
```

```
  signature *(`T <= num, T):T;
  signature /(`T <= num, T):T;
  signature /_float(`T <= num, T):T & float;
  implementation +(l@:num,r@:num):num;
  implementation -(l@:num,r@:num):num;
  implementation *(l@:num,r@:num):num;
  implementation /(l@:num,r@:num):num;
  implementation /_float(l@:num,r@:num):float;
  signature +(integer, float):float;
  signature +(float, integer):float;
  signature -(integer, float):float;
  signature -(float, integer):float;
  signature *(integer, float):float;
  signature *(float, integer):float;
  signature /(integer, float):float;
  signature /(float, integer):float;
  signature /_float(integer, float):float;
  signature /_float(float, integer):float;
  method negate(n:`T <= num):T;
  method -(n:`T <= num):T;
  method +(n:`T <= num):T;
  method -_ov(l@:num):num;
  method +_ov(l@:num,r@:num):num;
  method -_ov(l@:num,r@:num):num;
  method *_ov(l@:num,r@:num):num;
  method /_ov(l@:num,r@:num):num;
  method /_float_ov(l@:num,r@:num):num;
  method pred(i@num:`T <= num):T;
  method succ(i@num:`T <= num):T;
  method square(n:`T <= num):T;
  method cube(n:`T <= num):T;
  method abs(n:`T <= num):T;  -- absolute value
  method sign(x:num):int;  -- return -1, 0, or 1 depending on sign of argument
  method average(n1:num, n2:num):num;  -- arithmetic average: (n1+n2)/2
  method power(x@num:`T <= num, power:integer):S
                                     where signature *_ov(T,T):`S,
                                           S <= T;  -- exponentiation
  method **(x@num:`T <= num, power:integer):S
                                     where signature *_ov(T,T):`S, S <= T;  --
same as power
  method sqrt(x@num:`T <= num):T;
```

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.

2.3.1 Integers

In `integer.cecil`:

```
abstract object integer isa num;
method =(l@:integer,r@:integer):bool;
```

```
method <(l@:integer,r@:integer):bool;
implementation +(l@:integer,r@:integer):integer;
implementation -(l@:integer,r@:integer):integer;
implementation *(l@:integer,r@:integer):integer;
implementation /(l@:integer,r@:integer):integer;
method mod(l@:integer,r@integer:‘T <= integer):T;  -- modulus
signature %(integer, ‘T <= integer):T;  -- same as mod
implementation %(l@:integer,r@:integer):integer;
signature rem(integer, ‘T <= integer):T;  -- remainder
implementation rem(l@:integer,r@:integer):integer;
method -_ov(l@:integer):integer;
method +_ov(l@:integer,r@:integer):integer;
method -_ov(l@:integer,r@:integer):integer;
method *_ov(l@:integer,r@:integer):integer;
method /_ov(l@:integer,r@:integer):integer;
method %_ov(l@:integer,r@:integer):integer;
method <<(l@:integer,r@:integer):integer;  -- left shift
method >>(l@:integer,r@:integer):integer;  -- right arithmetic shift (extends sign)
method <<_ov(l@:integer,r@:integer):integer;
method >>_ov(l@:integer,r@:integer):integer;
method bit_and (l@:integer, r@:integer):integer;
method bit_or (l@:integer, r@:integer):integer;
method bit_xor (l@:integer, r@:integer):integer;
method bit_xnor (l@:integer, r@:integer):integer;
method bit_not(l@:integer):integer;
method get_bit(i@:integer, bit@:integer):integer;  -- extract ith bit (0 or 1)
method set_bit(i@:integer, bit@:integer):integer;  -- set ith bit to 1
method clear_bit(i@:integer, bit@:integer):integer;  -- set ith bit to 0
method as_int(l@integer:‘T <= integer):T;
method as_small_int(l@:integer):int;
signature as_small_int(integer, if_overflow:&():‘T):int|T;
method as_small_int_if_possible(l@:integer):integer;
method is_int8(val@:integer):bool;
method as_int8(val@:integer, if_overflow:&():integer):integer;
method log_base(x@:integer, base@:integer):int;  -- compute logarithm to nearest integer; rounds up
method exact_log_base(x@:integer, base@:integer,
                      if_not_exact:&():int):int;  -- compute logarithm, but invoke closure if integer result not exact
method is_even(i@:integer):bool;  -- test parity
method is_odd(i@:integer):bool;
method hash(i@:integer, range:‘T <= integer):S
    where signature %(integer, T):‘Q, signature abs(Q):‘S;
signature average(integer,integer):integer;
```

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.

```
method round_up(i:integer, nearest:integer):integer;
signature round_up(int, int):int;
method round_down(i:integer, nearest:integer):integer;
signature round_down(int, int):int;
```

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

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

```
method do_digits_increasing(i@:integer, c:&(digit:int,position:int):void):void;
method do_digits_increasing_base(i@:integer, base:int,
                                 c:&(digit:int,position:int):void):void;
method print_string(i@:integer):string;
method print_string_base(i@:integer, base:int):string;
method print_string_base(i@:int, base:int):string;
method print_string(i@:integer, len:int):string;
```

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

```
method parse_as_int(s@:string):integer;
method parse_as_int(s@:string, base@:int):integer;
method parse_as_int(s@:string, fail@closure:&():integer):integer;
method parse_as_int(s@:string, base@:int, fail:&():integer):integer;
method parse_as_small_int(s@:string):int;
method parse_as_small_int(s@:string, base@:int):int;
method parse_as_small_int(s@:string, fail@closure:&():int):int;
method parse_as_small_int(s@:string, base@:int, fail:&():int):int;
```

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.

In `small-int.cecil`:

```
extend int isa integer;
method = (l@:int, r@:int):bool;
method < (l@:int, r@:int):bool;
method <_unsigned (l@:int, r@:int):bool;
method <= (l@:int, r@:int):bool;
method <=_unsigned (l@:int, r@:int):bool;
method > (l@:int, r@:int):bool;
method >_unsigned (l@:int, r@:int):bool;
method >= (l@:int, r@:int):bool;
method >=_unsigned (l@:int, r@:int):bool;
method != (l@:int, r@:int):bool;
implementation + (l@:int, r@:int):int;
implementation - (l@:int, r@:int):int;
implementation * (l@:int, r@:int):int;
implementation / (l@:int, r@:int):int;
implementation % (l@:int, r@:int):int;
method negate_ov (l@:int, if_overflow:&():`T):int|T;
method add_ov (l@:int, r@:int, if_overflow:&():`T):int|T;
method sub_ov (l@:int, r@:int, if_overflow:&():`T):int|T;
method mul_ov (l@:int, r@:int, if_overflow:&():`T):int|T;
method div_ov (l@:int, r@:int, if_overflow:&():`T):int|T;
method mod_ov (l@:int, r@:int, if_overflow:&():`T):int|T;
```

```
method -_ov(l@:int):integer;
method +_ov (l@:int, r@:int):integer;
method -_ov (l@:int, r@:int):integer;
method *_ov (l@:int, r@:int):integer;
method /_ov (l@:int, r@:int):integer;
method %_ov (l@:int, r@:int):integer;
method as_single_float(l@:int):single_float;
method as_double_float(l@:int):double_float;
method bit_and (l@:int, r@:int):int;
method bit_or (l@:int, r@:int):int;
method bit_xor (l@:int, r@:int):int;
method bit_not(l@:int):int;
method << (l@:int, r@:int):int;
method >> (l@:int, r@:int):int;
method >>_logical (l@:int, r@:int):int;  -- right logical shift (no sign-extension)
method get_bit(i@:int, bit@:int):int;
method set_bit(i@:int, bit@:int):int;
method clear_bit(i@:int, bit@:int):int;
method is_even(i@:int):bool;
method sqrt(x@:int):int;
signature average(int,int):int;
signature pred(int):int;
signature succ(int):int;
method as_small_int(l@:int, if_overflow:&():`T):int;
```

Small integers (`int`) are the main representation of integers.

```
method do(count@:int, c:&(int):void):void;
method print_string(i@:int):string;
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
```

`do` on integers is a simple kind of for-loop. Expression `n.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.

In `big-int.cecil`:

```
template object big_int isa integer;
method =(l@:big_int, r@:big_int):bool;
method < (l@:big_int, r@:big_int):bool;
method <=(l@:big_int, r@:big_int):bool;
method >=(l@:big_int, r@:big_int):bool;
method > (l@:big_int, r@:big_int):bool;
method compare(l@:big_int, r@:big_int,
               if_less:&():`T, if_equal:&():`T, if_greater:&():`T
               ):T;
method -(l@:big_int):big_int;
method +(l@:big_int, r@:big_int):big_int;
method -(l@:big_int, r@:big_int):big_int;
method *(l@:big_int, r@:big_int):big_int;
method /(l@:big_int, r@:big_int):big_int;
method %(l@:big_int, r@:big_int):big_int;
```

```
method %(l@:big_int, r@:int):int;
method <<(l@:big_int, r@:integer):integer;
method >>(l@:big_int, r@:integer):integer;
method as_small_int(l@:big_int, if_overflow:&():`T):int|T;
signature as_big_int(integer):big_int;
implementation as_big_int(l@:big_int):big_int;
implementation as_big_int(l@:int):big_int;
method as_single_float(l@:big_int):single_float;
method as_double_float(l@:big_int):double_float;
method print_string_base(x@:big_int, base:int):string;
```

Big integers are an arbitrary-precision representation of integers.

2.3.2 Floating-point numbers

In `float.cecil`:

```
abstract object float isa num;
  method =(l@:float,r@:float):bool;
  method <(l@:float,r@:float):bool;
  method +(l@:float,r@:float):float;
  method -(l@:float,r@:float):float;
  method *(l@:float,r@:float):float;
  method /(l@:float,r@:float):float;
method as_float(f@:float):float;
method hash(f@:float, range:int):int;
signature sin(float):float;
signature cos(float):float;
signature tan(float):float;
signature asin(float):float;
signature acos(float):float;
signature atan(float):float;
signature exp(float):float;
signature log(float):float;
signature sqrt(float):float;
method log_base(x@:float, base@:float):float;  -- compute logarithm
signature round_as_int(float):int;
signature round(float):float;
signature round_towards_zero(float):float;
signature ceiling(float):float;
signature floor(float):float;
signature print_string(float, num_decimal_places:int):string;
signature print_string_full(float):string;
```

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.

In `single-float.cecil`:

```
extend single_float isa float;
method = (l@:single_float, r@:single_float):bool;
method < (l@:single_float, r@:single_float):bool;
```

```
method <= (l@:single_float, r@:single_float):bool;
method > (l@:single_float, r@:single_float):bool;
method >= (l@:single_float, r@:single_float):bool;
method != (l@:single_float, r@:single_float):bool;
method + (l@:single_float, r@:single_float):single_float;
method - (l@:single_float, r@:single_float):single_float;
method * (l@:single_float, r@:single_float):single_float;
method / (l@:single_float, r@:single_float):single_float;
method as_single_float(f@:single_float):single_float;
method as_double_float(f@:single_float):double_float;
method as_int(f@:single_float):int;
method sin(f@:single_float):single_float;
method cos(f@:single_float):single_float;
method tan(f@:single_float):single_float;
method asin(f@:single_float):single_float;
method acos(f@:single_float):single_float;
method atan(f@:single_float):single_float;
method exp(f@:single_float):single_float;
method log(f@:single_float):single_float;
method sqrt(f@:single_float):single_float;
method round_as_int(f@:single_float):int;
method round(f@:single_float):single_float;
method round_towards_zero(f@:single_float):single_float;
method ceiling(f@:single_float):single_float;
method floor(f@:single_float):single_float;
method print_string(f@:single_float):string;
method print_string(f@:single_float, num_decimal_places@:int):string;
method print_string_full(f@:single_float):string;
method parse_as_float(s@:string):single_float;
method parse_as_float(s@:string, if_error:&():single_float):single_float;
method parse_as_float(s@:vstring, if_error:&():single_float):single_float;
let pi:float;
```

In `double-float.cecil`:

```
extend double_float isa float;
method = (l@:double_float, r@:double_float):bool;
method < (l@:double_float, r@:double_float):bool;
method <= (l@:double_float, r@:double_float):bool;
method > (l@:double_float, r@:double_float):bool;
method >= (l@:double_float, r@:double_float):bool;
method != (l@:double_float, r@:double_float):bool;
method + (l@:double_float, r@:double_float):double_float;
method - (l@:double_float, r@:double_float):double_float;
method * (l@:double_float, r@:double_float):double_float;
method / (l@:double_float, r@:double_float):double_float;
method as_double_float(f@:double_float):double_float;
method as_single_float(f@:double_float):single_float;
method as_int(f@:double_float):int;
method sin(f@:double_float):double_float;
method cos(f@:double_float):double_float;
method tan(f@:double_float):double_float;
method asin(f@:double_float):double_float;
```

```
method acos(f@:double_float):double_float;
method atan(f@:double_float):double_float;
method exp(f@:double_float):double_float;
method log(f@:double_float):double_float;
method sqrt(f@:double_float):double_float;
method round_as_int(f@:double_float):int;
method round(f@:double_float):double_float;
method round_towards_zero(f@:double_float):double_float;
method ceiling(f@:double_float):double_float;  -- round towards positive infinity
method floor(f@:double_float):double_float;  -- round towards negative infinity
method print_string(f@:double_float):string;
method print_string(f@:double_float, num_decimal_places@:int):string;
method print_string_full(f@:double_float):string;
method parse_as_double(s@:string):double_float;
method parse_as_double(s@:string, if_error:&():double_float):double_float;
method parse_as_double(s@:vstring, if_error:&():double_float):double_float;
```

2.4 Characters

In `character.cecil`:

```
extend char isa ordered_hashable[char];
method from_ascii(i@:int):char;
method from_ascii(i@:int, if_error:&():char):char;
method ascii_code(c@:char):int;  -- convert between characters and integer ASCII codes
method =(l@:char,r@:char):bool;
method <(l@:char,r@:char):bool;
method <=(l@:char,r@:char):bool;
method is_lower_case(c@:char):bool;
method is_upper_case(c@:char):bool;
method is_letter(c@:char):bool;
method is_digit(c@:char):bool;
method is_digit(c@:char, base:int):bool;
method is_hex_digit(c@:char):bool;
method is_printable(c@:char):bool;
method hash(c@:char, range:int):int;
method as_string(c@:char):string;  -- convert to a string of a single character
method to_lower_case(c@:char):char;
method to_upper_case(c@:char):char;
method parse_as_int(c@:char):int;
method parse_as_int(c@:char, base:int):int;
method parse_as_int(c@:char, fail@closure:&():int):int;
method parse_as_int(c@:char, base:int, fail@closure:&():int):int;
method print_string(c@:char):string;
```

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

2.5 Object aggregation

In `pair.cecil`:

The `pair`, `triple`, and `quadruple` data structures can sometimes be useful, e.g., for returning a tuple of results from a function.

```
template object pair[T1,T2];
  extend type pair[`T1,`T2] subtypes pair[`S1 >= T1, `S2 >= T2];
  field first (@:pair[`T1,`T2]):T1;
  field second(@:pair[`T1,`T2]):T2;
  method pair(x:`T1, y:`T2):pair[T1,T2];
  method print_string(p@:pair[`T1,`T2]):string;
extend pair[`T1 <= comparable[T1], `T2 <= comparable[T2]]
        isa comparable[pair[T1,T2]];
  method =(p1@:pair[`T1 <= comparable[T1], `T2 <= comparable[T2]],
           p2@:pair[T1, T2]):bool;
extend pair[`T1 <= partially_ordered[T1], `T2 <= partially_ordered[T2]]
        isa partially_ordered[pair[T1,T2]];
  method <=(p1@:pair[`T1 <= partially_ordered[T1],
                     `T2 <= partially_ordered[T2]],
            p2@:pair[T1, T2]):bool;
  method <(p1@:pair[`T1 <= partially_ordered[T1],
                    `T2 <= partially_ordered[T2]],
           p2@:pair[T1, T2]):bool;
extend pair[`T1 <= hashable[T1], `T2 <= hashable[T2]] isa hashable[pair[T1,T2]];
  method hash(p@:pair[`T1 <= hashable[T1], `T2 <= hashable[T2]],
              range:int):int;
```

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, triples, and quadruples do not start with `new_` – like `cons`, they are exceptions from our naming convention for constructor methods.)

```
template object triple[T1,T2,T3];
  extend type triple[`T1,`T2,`T3]
        subtypes triple[`S1 >= T1, `S2 >= T2, `S3 >= T3];
  field signature first (triple[`T1,`T2,`T3]):T1;
  field signature second(triple[`T1,`T2,`T3]):T2;
  field third(@:triple[`T1,`T2,`T3]):T3;
  method triple(x:`T1, y:`T2, z:`T3):triple[T1,T2,T3];
  method print_string(p@:triple[`T1,`T2,`T3]):string;
extend triple[`T1 <= comparable[T1], `T2 <= comparable[T2],
              `T3 <= comparable[T3]]
        isa comparable[triple[T1,T2,T3]];
  method =(p1@:triple[`T1 <= comparable[T1],
                      `T2 <= comparable[T2],
                      `T3 <= comparable[T3]],
           p2@:triple[T1, T2, T3]):bool;
extend triple[`T1 <= partially_ordered[T1], `T2 <= partially_ordered[T2],
              `T3 <= partially_ordered[T3]]
        isa partially_ordered[triple[T1,T2,T3]];
  method <=(p1@:triple[`T1 <= partially_ordered[T1],
                       `T2 <= partially_ordered[T2],
                       `T3 <= partially_ordered[T3]],
```

```
            p2@:triple[T1, T2, T3]):bool;
extend triple[`T1 <= hashable[T1], `T2 <= hashable[T2], `T3 <= hashable[T3]]
        isa hashable[triple[T1,T2,T3]];
  method hash(p@:triple[`T1 <= hashable[T1],
                        `T2 <= hashable[T2],
                        `T3 <= hashable[T3]],
              range:int):int;
```

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.

```
template object quadruple[T1,T2,T3,T4];
  extend type quadruple[`T1,`T2,`T3,`T4]
        subtypes quadruple[`S1 >= T1, `S2 >= T2, `S3 >= T3, `S4 >= T4];
  field signature first (quadruple[`T1,`T2,`T3,`T4]):T1;
  field signature second(quadruple[`T1,`T2,`T3,`T4]):T2;
  field signature third (quadruple[`T1,`T2,`T3,`T4]):T3;
  field fourth(@:quadruple[`T1,`T2,`T3,`T4]):T4;
  method quadruple(x:`T1, y:`T2, z:`T3, w:`T4):quadruple[T1,T2,T3,T4];
  method print_string(p@:quadruple[`T1,`T2,`T3,`T4]):string;
extend quadruple[`T1 <= comparable[T1], `T2 <= comparable[T2],
                 `T3 <= comparable[T3], `T4 <= comparable[T4]]
        isa comparable[quadruple[T1,T2,T3,T4]];
  method =(p1@:quadruple[`T1 <= comparable[T1],
                         `T2 <= comparable[T2],
                         `T3 <= comparable[T3],
                         `T4 <= comparable[T4]],
           p2@:quadruple[T1, T2, T3, T4]):bool;
extend quadruple[`T1 <= partially_ordered[T1], `T2 <= partially_ordered[T2],
                 `T3 <= partially_ordered[T3], `T4 <= partially_ordered[T4]]
        isa partially_ordered[quadruple[T1,T2,T3,T4]];
  method <=(p1@:quadruple[`T1 <= partially_ordered[T1],
                          `T2 <= partially_ordered[T2],
                          `T3 <= partially_ordered[T3],
                          `T4 <= partially_ordered[T4]],
            p2@:quadruple[T1, T2, T3, T4]):bool;
extend quadruple[`T1 <= hashable[T1], `T2 <= hashable[T2],
                 `T3 <= hashable[T3], `T4 <= hashable[T4]]
        isa hashable[quadruple[T1,T2,T3,T4]];
  method hash(p@:quadruple[`T1 <= hashable[T1], `T2 <= hashable[T2],
                           `T3 <= hashable[T3], `T4 <= hashable[T4]],
              range:int):int;
```

Quadruples are also provided.

3 Control structures

3.1 Booleans and branching

In `boolean.cecil`:

```
abstract object bool isa hashable[bool]; -- hashable implies comparable
concrete representation true isa bool;
concrete representation false isa bool;
```

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:

```
signature if(bool, tc:&():`T, fc:&():`T):T;
method if_false(t@:bool, fc:&():`T, tc:&():`T):T;
method if(t@:bool, tc:&():void):void;
method if_false(t@:bool, fc:&():void):void;
method =(l@:bool, r@:bool):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. `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 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.

```
signature |(bool, bool):bool;
signature |(bool, &():bool):bool;
signature &(bool, bool):bool;
signature &(bool, &():bool):bool;
signature not(bool):bool;
method =>(l@:bool, r@:bool):bool;
method =>(l@:bool, r@closure:&():bool):bool;
```

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 }, {
    ...
});
```

```
method assert(b@:bool):void;
method assert(b@:bool, msg@:string):void;
method assert(b@:bool, msg_closure@closure:&():string):void;
```

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.

```
signature as_integer(bool):int;
method hash(b@:bool, range:int):int;
method if(@:true, tc:&():‘T, fc:&():‘T):T;
method |(@:true, @:bool):bool;
method |(@:true, @closure:&():bool):bool;
method &(@:true, r@:bool):bool;
method &(@:true, r@closure:&():bool):bool;
method not(@:true):bool;
method as_integer(@:true):int;
method print_string(@:true):string;
method if(@:false, tc:&():‘T, fc:&():‘T):T;
method |(@:false, r@:bool):bool;
method |(@:false, r@closure:&():bool):bool;
method &(@:false, @:bool):bool;
method &(@:false, @closure:&():bool):bool;
method not(@:false):bool;
method as_integer(@:false):int;
method print_string(@:false):string;
```

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

3.2 Looping and closures

In `closure.cecil`:

```
abstract representation closure;
```

The `closure` representation is the parent of all closure expressions. Many control structures are defined on this object. But it's a regular object, so other user-defined objects can be descendants (e.g., see `eval.cecil` in the compiler directory tree).

```
method loop(c@closure:&():void):none;
```

The `loop` method 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 method.

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

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

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

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

Other while-do and do-until loops are implemented using the `{while,until}[_{true,false}]` methods. The one-argument `while_{true,false}` methods 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.

```
method exit(c@closure:&(exit:&():none):void):void;
method exit_value(c:&(exit:&(`T):none):`T):T;
method exit_continue(c:&(exit:&():none, continue:&():none):void):void;
method exit_value_continue(c:&(exit:&(`T):none, continue:&():none):T):T;
```

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.

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

For loops, some additional methods are defined for convenience.

The `loop[_exit[_value]][_continue]` methods evaluate the `body` closure again and again until the `break` closure is evaluated inside body. For the `_continue` version, execution of body can be restarted from the beginning by evaluating the `continue` closure. The `_value` version returns 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 `do`, two `exit` methods should be used, as in the following example. The outer method encloses the iterator and provides breaking out of the loop, while the inner method encloses the loop body and provides continuing to the next iteration:

```
exit(&(break:&():none){
    10.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
            ...
        });
    });
});
```

```
template object case_pair[T];
  extend type case_pair['T] subtypes case_pair['S >= T];
  field cond(@:case_pair['T]):&():bool;
  field stmt(@:case_pair['T]):&():T;
method case(c:&():bool, s:&():'T):case_pair[T];
method else(s:&():'T):case_pair[T];
method switch(t@:ordered_collection[case_pair['T]]):T;
method unrolled_switch(t@:i_vector[case_pair['T]]):T;
```

Case statements are supported through the `switch`, `case`, and `else` methods. 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` method. (Thus `switch` is very much like Lisp's `cond`.) The `switch` method dies with a run-time error if none of the cases match 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 Cecil control structures, the `switch` construct is not as efficient as the 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. (Recently, 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.)

There's an inconsistency between the closure representation and closure types, however. The closure representation is not parameterized, but closure types (using the `&(...):...` syntax) effectively are parameterized by the closure's argument and result types. The closure representation should be parameterized, too. The language defines the standard contravariant subtyping relationship among closure types, i.e.:

```
extend &(T1, ..., TN):T subtypes &(`S1 >= T1, ..., `SN >= TN):(`S <= T)
```

3.3 Exception handling

A couple of methods are defined on closures to specialize error and exception handling.

In `error.cecil`:

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

The `handle_system_errors` method executes its argument closure. Any Cecil errors (e.g., `error` or `exit` calls) caused during evaluation of the closure are caught and suppressed. The boolean return value indicates whether an error occurred.

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

The `unwind_protect` method allows a "clean up" closure to be executed whenever control returns through the `unwind_protect` method call, either normally, via a non-local return, or via a Cecil error. The `cl` closure is invoked. When control returns from this invocation, the `on_return` closure is invoked. When this closure completes, the returning of the `cl` closure is resumed. If the `cl` closure returned normally, the result of this closure is returned as the result of the `unwind_protect method`; otherwise it continues to throw whatever exception the body did. (`unwind_protect` is similar to the like-named construct in Common Lisp.)

4 Collections

In `collection.cecil`:

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: generic, immutable, and mutable, with the latter two indicated by `i_` and `m_` prefixes. For example:

```
abstract object indexed[`T] isa table[int,T], indexed[`S >= T];
abstract object i_indexed[`T] isa indexed[T];
abstract object m_indexed[`T] isa indexed[T];
```

This troika of versions of each abstraction is a standard idiom among the collection classes. A generic base class defines the abstract read-only behavior of the object, but doesn't specify whether the collection is mutable. One subclass adds the immutability specification, and another subclass adds the mutator operations. An immutable collection may contain mutable objects that are side-effected while in the collection, but the collection itself cannot be changed.

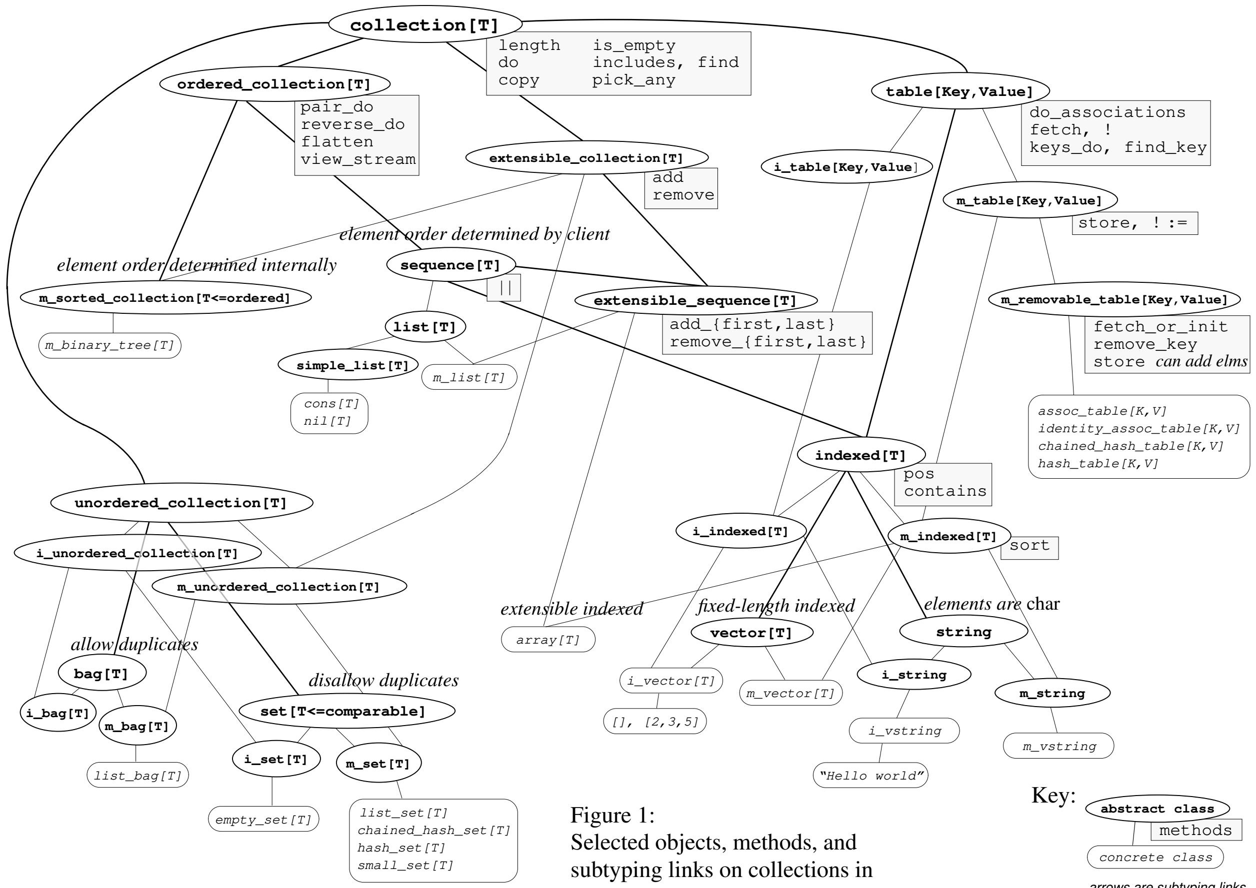

Figure 1:
Selected objects, methods, and subtyping links on collections in the standard library.

There are three 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]` objects, which may be inherited by mutable objects.

Separating the read-only from the read-write interfaces (e.g., `unordered_collection` vs. `m_unordered_collection`) supports useful subtyping relationships among the read-only interfaces. To support the most reuse, the generic read-only interface should be used as a type declaration whenever possible. Only if mutation is required should the mutable subtype interface be used. The immutable interface must be distinguished from the generic read-only interface because the mutable interface cannot be a subtype of the immutable interface, considering the behavior specification implied by the immutable interface.

For a given abstraction, e.g., `indexed`, both the immutable and mutable versions containing elements of type `T` (or subtype of `T`) subclass the generic version with elements of type `T` or any supertype of `T`. In addition, the immutable version, `i_indexed[T]`, is a subtype of any immutable collection (of the same kind) of a supertype of `T`:

```
extend type i_indexed[`T] subtypes i_indexed[`S >= T];
```

In contrast, a mutable version of a type `T` has no subtyping relation to a mutable collection with a different element type, e.g., `m_indexed[int]` is unrelated to `m_indexed[num]` : `m_indexed[int]` cannot have floats stored in it (unlike `m_indexed[num]`), and so `m_indexed[int]` is not a subtype of `m_indexed[num]`. Also, `m_indexed[num]` can contain things other than `int`s and reveal this through `do`, `pick_any`, etc. (unlike `m_indexed[int]`), so the reverse subtyping relation doesn't hold, either.

Finally, the immutable and mutable versions subclass those of the abstraction higher in the class hierarchy:

```
extend i_indexed[`T] isa i_table[int,T];
extend m_indexed[`T] isa m_table[int,T];
```

4.1 Basic collections

```
abstract object collection[T];
extend type collection[`T] subtypes collection[`S >= T];
```

`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`.

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

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

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

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
});
```

```
method do_allowing_updates(t@: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.)

```
method includes(c@:collection[‘T <= comparable[T]], x:T):bool;
method includes_all(c1@:collection[‘T <= comparable[T]],
                    c2:collection[T]):bool;
method includes_some(c@:collection[‘T], test:&(T):bool):bool;
```

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`.

```
method count(c@:collection[‘T <= comparable[T]], x:T):int;
```

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

```
method count_pred(c@:collection[‘T <= comparable[T]], test:&(T):bool):int;
```

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

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

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

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

Tests whether the predicate is true of `every` or `any` collection element.

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

Implements the classic functional reduce operation over collections.

```
method reduce(t@:collection[‘T], bin_op:&(T,T):T):T;
method reduce_nonempty(t@:collection[‘T], bin_op:&(T,T):T):T;
method reduce_nonempty(t@:collection[‘T], bin_op:&(T,T):T, if_empty:&():T):T;
```

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

```
method min(t@:collection[`T <= ordered[T]]):T;
method min_over_all(t@:collection[`T <= ordered[T]]):T;
method min_over_all(t@:collection[`T <= ordered[T]], if_empty:&():T):T;
method max(t@:collection[`T <= ordered[T]]):T;
method max_over_all(t@:collection[`T <= ordered[T]]):T;
method max_over_all(t@:collection[`T <= ordered[T]], if_empty:&():T):T;
method average(t@:collection[`T <= num]):T;
method average_over_all(t@:collection[`T <= num]):T;
method average_over_all(t@:collection[`T <= num], if_empty:&():T):T;
```

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.

```
method pick_any(c@:collection[`T]):T;
method pick_any(c@:collection[`T1], if_empty:&():`T2):T1|T2;
method only(c@:collection[`T]):T;
method only(c@:collection[`T1], if_non_singleton:&():`T2):T1|T2;
```

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.

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

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.

```
method as_ordered_collection(c@:collection[`T]):ordered_collection[T];
method as_m_indexed(c@:collection[`T]):m_indexed[T];
method as_vector(c@:collection[`T]):vector[T];
method as_i_vector(c@:collection[`T]):i_vector[T];
method as_m_vector(c@:collection[`T]):m_vector[T];
method as_byte_vector(c@:collection[int]):byte_vector;
method as_i_byte_vector(c@:collection[int]):i_byte_vector;
method as_m_byte_vector(c@:collection[int]):m_byte_vector;
method as_short_vector(c@:collection[int]):short_vector;
method as_i_short_vector(c@:collection[int]):i_short_vector;
method as_m_short_vector(c@:collection[int]):m_short_vector;
method as_list_set(c@:collection[`T <= comparable[T]]):m_set[T];
```

Collection conversions may or may not involve a copy, e.g., `as_vector` on a `list` copies, while `as_vector` on a `vector` does not.

```
method print_string(c@:collection[`T]):string;
method print(c@:collection[`T]):void;
signature collection_name(collection[`T]):string;
method open_brace(c@:collection[`T]):string;
method close_brace(@:collection[`T]):string;
method elems_print_string(c@:collection[`T]):string;
```

```
method elems_print(c@:collection[`T]):void;
method elem_separator(@:collection[`T]):string;
method elem_print_string(t@:collection[`T], elem:T, first:bool):string;
method elem_print(t@:collection[`T], elem:T, first:bool):void;
```

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.

4.2 Unordered collections

In `unordered.cecil`:

```
abstract object unordered_collection[T] isa collection[T];
  extend type unordered_collection[`T] subtypes unordered_collection[`S >= T];
  method count(c@:unordered_collection[`T <= comparable[T]], x:T):int;
```

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 unordered_collection[`T <= comparable[T]]
              isa comparable[unordered_collection[T]];
  method =(c1@:unordered_collection[`T <= comparable[T]],
           c2@:unordered_collection[T]):bool;
  extend unordered_collection[`T <= hashable[T]]
              isa hashable[unordered_collection[T]];
  method hash(c@:unordered_collection[`T <= hashable[T]], range:int):int;
```

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.

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

```
abstract object i_unordered_collection[T] isa unordered_collection[T];
  extend type i_unordered_collection[`T]
              subtypes i_unordered_collection[`S >= T];
  method copy(c@:i_unordered_collection[`T]):i_unordered_collection[T];
  method copy_empty(c@:i_unordered_collection[`T]
                   ):i_unordered_collection[T];
```

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 object m_unordered_collection[T]
       isa unordered_collection[T], extensible_collection[T];
  method copy(c@:m_unordered_collection[`T]):m_unordered_collection[T];
  signature copy_empty(m_unordered_collection[`T]):m_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.

4.2.1 Bags

In `bag.cecil`:

```
abstract object bag[T] isa unordered_collection[T];
extend type bag[`T] subtypes bag[`S >= T];
signature copy(bag[`T]):bag[T];
abstract object i_bag[T] isa bag[T], i_unordered_collection[T];
signature copy(i_bag[`T]):i_bag[T];
abstract object m_bag[T] isa bag[T], m_unordered_collection[T];
extend m_bag[`T <= comparable[T]] isa removable_collection[T];
signature copy(m_bag[`T]):m_bag[T];
```

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

```
template object list_bag[T] isa m_bag[T];
  method collection_name(@:list_bag[`T]):string;
  method length(m@:list_bag[`T]):int;
  method is_empty(m@:list_bag[`T]):bool;
  method do(m@:list_bag[`T], c:&(T):void):void;
  method add(m@:list_bag[`T], x:T):void;
  method remove(m@:list_bag[`T <= comparable[T]], x:T,
                if_absent:&():void):void;
  method remove_any(m@:list_bag[`T], if_empty:&():`S):T|S;
  method remove_if(m@:list_bag[`T], pred:&(T):bool):int;
  method remove_all(m@:list_bag[`T]):void;
  method new_list_bag[T]():list_bag[T];
  method copy_empty(c@:list_bag[`T]):list_bag[T];
  method as_ordered_collection(c@:list_bag[`T]):m_list[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.

4.2.2 Sets

In `set.cecil`:

```
abstract object set[T <= comparable[T]] isa unordered_collection[T];
extend type set[`T] subtypes set[`S >= T];
```

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

```
method union(m1@:set[`T], m2@:set[T]):m_set[T];
method intersection(m1@:set[`T], m2@:set[T]):m_set[T];
method difference(m1@:set[`T], m2@:set[T]):m_set[T];
method is_disjoint(m1@:set[`T], m2@:set[T]):bool;
method overlaps(m1@:set[`T], m2@:set[T]):bool;
method is_subset(m1@:set[`T], m2@:set[T]):bool;
```

```
method count(c@:set[`T], x:T):int;
method collection_name(@:set[`T]):string;
signature copy(set[`T]):set[T];
abstract object i_set[T <= comparable[T]] isa set[T], i_unordered_collection[T];
method collection_name(@:i_set[`T]):string;
```

Sets support a collection of functional set operations which, given argument sets, return a new set.

```
concrete object empty_set[T <= comparable[T]] isa i_set[T];
  method print_string(@:empty_set[`T]):string;
  method copy(s@:empty_set[`T]):empty_set[T];
  method copy_empty(s@:empty_set[`T]):empty_set[T];
  method do(s@:empty_set[`T], c:&(T):void):void;
  method do_allowing_updates(s@:empty_set[`T], c:&(T):void):void;
  signature do_allowing_updates(empty_set[`T]|m_set[`T], c:&(T):void):void;
  method length(@:empty_set[`T]):int;
  method is_empty(@:empty_set[`T]):bool;
```

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

```
abstract object m_set[T <= comparable[T]] isa set[T],
                                              m_unordered_collection[T],
                                              removable_collection[T];
method collection_name(@:m_set[`T]):string;
method add(m@:m_set[`T], x:T):void;
method check_if_missing_and_add(m@:m_set[`T], x:T):bool;
signature add_functional(m_set[`T], x:T):m_set[T];
signature copy_empty(m_set[`T]):m_set[T];
method copy(c@:m_set[`T]):m_set[T];
```

Mutable sets can be added to.

4.2.3 Set implementations

```
template object list_set[T <= comparable[T]] isa m_set[T];
method collection_name(@:list_set[`T]):string;
method add(m@:list_set[`T], x:T):void;
method add_nonmember(m@:list_set[`T], x:T):void;
method new_list_set[T <= comparable[T]]():list_set[T];
method copy_empty(c@:list_set[`T]):list_set[T];
```

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

```
template object chained_hash_set[T <= hashable[T]] isa m_set[T];
method collection_name(@:chained_hash_set[`T]):string;
method do(t@:chained_hash_set[`T], c:&(T):void):void;
method do_allowing_updates(t@:chained_hash_set[`T], c:&(T):void):void;
method includes(t@:chained_hash_set[`T], x:T):bool;
method add(t@:chained_hash_set[`T], x:T):void;
method add_nonmember(t@:chained_hash_set[`T], x:T):void;
method remove(t@:chained_hash_set[`T], x:T, if_absent:&():void):void;
method remove_if(t@:chained_hash_set[`T], pred:&(T):bool):int;
```

```
method remove_any(t@:chained_hash_set[`T], if_empty:&():`S):T|S;
method remove_all(t@:chained_hash_set[`T]):void;
method union(m1@:chained_hash_set[`T], m2@:chained_hash_set[T]):m_set[T];
method intersection(m1@:chained_hash_set[`T],
                    m2@:chained_hash_set[T]):m_set[T];
let var default_chained_hash_set_size:int;
method new_chained_hash_set[T <= hashable[T]]():chained_hash_set[T];
method new_chained_hash_set[T <= hashable[T]](size:int):chained_hash_set[T];
method copy_empty(c@:chained_hash_set[`T]):chained_hash_set[T];
```

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

In `hash-set.cecil`:

```
template object hash_set[T <= hashable[T]] isa m_set[T], open_table[T];
method collection_name(@:hash_set[`T]):string;
method do(t@:hash_set[`T], c:&(T):void):void;
method do_allowing_updates(t@:hash_set[`T], c:&(T):void):void;
method includes(t@:hash_set[`T], x:T):bool;
method add(t@:hash_set[`T], x:T):void;
method remove(t@:hash_set[`T], x:T, if_absent:&():void):void;
method remove_if(t@:hash_set[`T], pred:&(T):bool):int;
method remove_any(t@:hash_set[`T], if_empty:&():`S):T|S;
method remove_all(t@:hash_set[`T]):void;
method union(m1@:hash_set[`T], m2@:hash_set[T]):m_set[T];
method intersection(m1@:hash_set[`T], m2@:hash_set[T]):m_set[T];
method new_hash_set[T <= hashable[T]]():hash_set[T];
method new_hash_set[T <= hashable[T]](size:int):hash_set[T];
method copy_empty(c@:hash_set[`T]):hash_set[T];
method copy(t@:hash_set[`T]):hash_set[T];
```

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.

In `small-set.cecil`:

```
concrete object empty_big_set[T <= hashable[T]]
        isa i_unordered_collection[T],
            functionally_extensible_removable_collection[T];
  method length(@:empty_big_set[`T]):int;
  method do(@:empty_big_set[`T], c:&(T):void):void;
  method includes(@:empty_big_set[`T], x:T):bool;
  method add_functional(@:empty_big_set[`T], x:T):m_set[T];
  method remove(@:empty_big_set[`T], x:T, if_absent:&():void):void;
  method remove_any(@:empty_big_set[`T], if_empty:&():`S):T|S;
  method remove_all(@:empty_big_set[`T]):void;
  method copy_empty(t@:empty_big_set[`T]):empty_big_set[T];
```

```
  method copy(t@:empty_big_set[`T]):empty_big_set[T];
  method print_string(t@:empty_big_set[`T]):string;
  method collection_name(@:empty_big_set[`T]):string;
template object small_set[T <= hashable[T]] isa m_set[T];
method collection_name(@:small_set[`T]):string;
method length(t@:small_set[`T]):int;
method is_empty(t@:small_set[`T]):bool;
method includes(t@:small_set[`T], x:T):bool;
method add_nonmember(t@:small_set[`T], x:T):void;
method remove(t@:small_set[`T], x:T, if_absent:&():void):void;
method remove_any(t@:small_set[`T], if_empty:&():`S):T|S;
method remove_all(t@:small_set[`T]):void;
method shrink_set(t@:small_set[`T]):void;
method do(t@:small_set[`T], c:&(T):void):void;
method new_small_set[T <= hashable[T]]():small_set[T];
method copy_empty(t@:small_set[`T]):small_set[T];
method copy(t@:small_set[`T]):small_set[T];
```

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`.

In `bit-set.cecil`:

```
abstract object bit_set[T <= comparable[T]] isa m_set[T], hashable[bit_set[T]];
  signature element_to_index(bit_set[`T], T):int;
  signature index_to_element(bit_set[`T], int):T;
  signature new_bit_set(bit_set[`T], bit_vector, cached_length:int):bit_set[T];
  method hash(a@:bit_set[`T], range:int):int;
  method copy(a@bit_set[`T]:`S <= bit_set[`T]):R
                      where signature new_bit_set(S, bit_vector, int):`R;
  method union(a@bit_set[`T]:`S <= bit_set[`T], b@bit_set[`T]:`S):R
                      where signature new_bit_set(S, bit_vector, int):`R;
  method intersection(a@bit_set[`T]:`S <= bit_set[`T], b@bit_set[`T]:`S):R
                      where signature new_bit_set(S, bit_vector, int):`R;
  method difference(a@bit_set[`T]:`S <= bit_set[`T], b@bit_set[`T]:`S):R
                      where signature new_bit_set(S, bit_vector, int):`R;
  method union_in_place(a@bit_set[`T]:`S <= bit_set[`T],
                      b@bit_set[`T]:`S):void;
  method intersection_in_place(a@bit_set[`T]:`S <= bit_set[`T],
                             b@bit_set[`T]:`S):void;
  method difference_in_place(a@bit_set[`T]:`S <= bit_set[`T],
                           b@bit_set[`T]:`S):void;
  method is_empty(a@:bit_set[`T]):bool;
  method =(a@:bit_set[`T], b@:bit_set[T]):bool;
  method remove(a@:bit_set[`T], elem:T, if_absent:&():void):void;
  method includes(a@:bit_set[`T], elem:T):bool;
  method includes_id(a@:bit_set[`T], idx:int):bool;
  method includes_all(a@:bit_set[`T], b@:bit_set[T]):bool;
  method add(a@:bit_set[`T], elem:T):void;
  method add_id(a@:bit_set[`T], idx:int):void;
  method add_all(a@:bit_set[`T], b@:bit_set[T]):void;
  method do(a@:bit_set[`T], cl:&(T):void):void;
```

```
  method is_disjoint(m1@:bit_set[`T], m2@:bit_set[T]):bool;
  method remove_any(a@:bit_set[`T], if_empty:&():`S):T|S;
  method remove_all(a@:bit_set[`T]):void;
  method length(a@:bit_set[`T]):int;
  method collection_name(@:bit_set[`T]):string;
abstract object caching_bit_set_element[T <= hashable[T]]
                                isa hashable[caching_bit_set_element[T]]
                                -- type parameter is contravariant:
                                subtypes caching_bit_set_element[`S <= T];
  signature id_manager(caching_bit_set_element[`T]
                      ):bit_set_id_manager[caching_bit_set_element[T]];
  var field id_num(t@:caching_bit_set_element[`T]):int;
  method reset_id_num(t@:caching_bit_set_element[`T]):void;
abstract object caching_bit_set[T <= caching_bit_set_element[T]] isa bit_set[T];
  signature id_manager(caching_bit_set[`T]):bit_set_id_manager[T];
  method element_to_index(t@:caching_bit_set[`T], elem:T):int;
  method index_to_element(t@:caching_bit_set[`T], idx:int):T;
abstract object caching_bit_set_element_2[T <= hashable[T]]
                                isa hashable[caching_bit_set_element_2[T]]
                                -- type parameter is contravariant:
                                subtypes caching_bit_set_element_2[`S <= T];
  signature id_manager_2(caching_bit_set_element_2[`T]
                        ):bit_set_id_manager[caching_bit_set_element_2[T]];
  var field id_num_2(t@:caching_bit_set_element_2[`T]):int;
  method reset_id_num_2(t@:caching_bit_set_element_2[`T]):void;
abstract object caching_bit_set_2[T <= caching_bit_set_element_2[T]]
                                isa bit_set[T];
  signature id_manager(caching_bit_set_2[`T]):bit_set_id_manager[T];
  method element_to_index(t@:caching_bit_set_2[`T], elem:T):int;
  method index_to_element(t@:caching_bit_set_2[`T], idx:int):T;
template object bit_set_id_manager[T <= hashable[T]];
  method element_to_index(t@:bit_set_id_manager[`T], elem:T):int;
  method index_to_element(t@:bit_set_id_manager[`T], idx:int):T;
  method reset_id_manager(t@:bit_set_id_manager[`T]):void;
  method length(t@:bit_set_id_manager[`T]):int;
  method new_bit_set_id_manager[T <= hashable[T]]()
                :bit_set_id_manager[T];
abstract object hashing_bit_set[T <= hashable[T]] isa bit_set[T];
  signature id_manager(hashing_bit_set[`T]
                      ):hashing_bit_set_id_manager[T];
  method element_to_index(t@:hashing_bit_set[`T], elem:T):int;
  method index_to_element(t@:hashing_bit_set[`T], idx:int):T;
template object hashing_bit_set_id_manager[T <= hashable[T]]
                                                isa bit_set_id_manager[T];
  method element_to_index(t@:hashing_bit_set_id_manager[`T], elem:T):int;
  method reset_id_manager(t@:hashing_bit_set_id_manager[`T]):void;
  method new_hashing_bit_set_id_manager[T <= hashable[T]]()
                :hashing_bit_set_id_manager[T];
```

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.

4.3 Ordered collections and sequences

In `ordered.cecil`:

```
abstract object ordered_collection[T] isa collection[T];
  extend type ordered_collection[`T] subtypes ordered_collection[`S >= T];
```

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.

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

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

```
signature reverse_do(ordered_collection[`T], cl:&(T):void):void;
method reverse(s@:ordered_collection[`T]):sequence[T];
```

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

```
extend ordered_collection[`T <= comparable[T]]
                isa comparable[ordered_collection[T]];
method =(c1@:ordered_collection[`T <= comparable[T]],
         c2@:ordered_collection[T]):bool;
extend ordered_collection[`T <= ordered[T]]
                isa ordered_using_compare[ordered_collection[T]];
method compare(c1@:ordered_collection[`T <= ordered[T]],
               c2@:ordered_collection[`T <= ordered[T]],
               if_less:&():`S, if_equal:&():`S, if_greater:&():`S
               ):S;
extend ordered_collection[`T <= hashable[T]]
                isa hashable[ordered_collection[T]];
extend ordered_collection[`T <= ordered_hashable[T]]
                isa ordered_hashable[ordered_collection[T]];
let hash_shift:int;
let num_hash_bits:int;
method hash(c@:ordered_collection[`T <= hashable[T]], range:int):int;
```

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.

```
method as_ordered_collection(c@:ordered_collection[`T]):ordered_collection[T];
method as_vstring(c@:ordered_collection[char]):vstring;
method as_m_vstring(c@:ordered_collection[char]):m_vstring;
signature view_stream(ordered_collection[`T]):stream[T];
signature copy(ordered_collection[`T]):ordered_collection[T];
```

Ordered collections of characters can be converted to **vstring** or **m_vstring** representations, and any ordered collection can be viewed as a stream. (In the future, we plan to build a more general mechanism for converting collections to different representations.)

```
method fetch(a@:ordered_collection[`T], i:int):T;
method !(a@:ordered_collection[`T], i:int):T;
method first (a@:ordered_collection[`T]):T;
method second(a@:ordered_collection[`T]):T;
method third (a@:ordered_collection[`T]):T;
method fourth(a@:ordered_collection[`T]):T;
method next_to_last(a@:ordered_collection[`T]):T;
method last  (a@:ordered_collection[`T]):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.

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

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.

In sequence.cecil:

```
abstract object sequence[T] isa ordered_collection[T];
extend type sequence[`T] subtypes sequence[`S >= T];
```

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.)

```
method ||(s1@:sequence[`T], s2@:sequence[`T]):sequence[T];
signature copy(sequence[`T]):sequence[T];
```

Concatenating two sequences always produces a new sequence

4.4 Adding and removing elements

In `extensible.cecil`:

```
abstract object removable_collection[T] isa collection[T];
signature remove(removable_collection[`T], x:T, if_absent:&():void):void;
method remove(c@:removable_collection[`T], x:T):void;
method remove_if(c@:removable_collection[`T], pred:&(T):bool):int;
signature remove_any(removable_collection[`T], if_empty:&():`S):T|S;
method remove_any(c@:removable_collection[`T]):T;
method remove_all(c@:removable_collection[`T]):void;
```

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.

```
abstract object functionally_extensible_collection[T] isa collection[T];
  signature add_functional(functionally_extensible_collection[`T], x:T
                           ):functionally_extensible_collection[T];
```

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.

```
abstract object functionally_extensible_removable_collection[T]
        isa removable_collection[T], functionally_extensible_collection[T];
  signature add_functional(functionally_extensible_removable_collection[`T],x:T
                           ):functionally_extensible_removable_collection[T];
  signature copy(functionally_extensible_removable_collection[`T]
                 ):functionally_extensible_removable_collection[T];
abstract object extensible_collection[T]
        isa functionally_extensible_removable_collection[T];
signature add(extensible_collection[`T], x:T):void;
method add_nonmember(c@:extensible_collection[`T], x:T):void;
method add_all(c@:extensible_collection[`T], xs:collection[T]):void;
method add_all_nonmember(c@:extensible_collection[`T], xs:collection[T]):void;
method add_functional(m@:extensible_collection[`T], x:T
                      ):extensible_collection[T];
```

A blend of `functionally_extensible_collection` and `removable_collection`.

```
abstract object extensible_ordered[T]
                isa extensible_collection[T], ordered_collection[T];
signature remove_first(extensible_ordered[`T], if_empty:&():`S):T|S;
method remove_first(c@:extensible_ordered[`T]):T;
signature remove_last(extensible_ordered[`T], if_empty:&():`S):T|S;
method remove_last(c@:extensible_ordered[`T]):T;
method remove_any(c@:extensible_ordered[`T], if_empty:&():`S):T|S;
```

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

```
abstract object extensible_sequence[T] isa extensible_ordered[T], sequence[T];
signature add_first(extensible_sequence[`T], x:T):void;
signature add_last (extensible_sequence[`T], x:T):void;
method add(c@:extensible_sequence[`T], x:T):void;
method add_all_last(s@:extensible_sequence[`T], c@:ordered_collection[T]):void;
```

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

4.5 Tables (mappings)

In `table.cecil`:

```
abstract object table[Key, Value] isa collection[Value];
extend type table[`Key,`Value] subtypes table[Key, `Value1 >= Value];
```

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.

```
signature do_associations(table[`Key,`Value], &(Key,Value):void):void;
method do_associations_allowing_updates(t@:table[`Key,`Value],
                                         c:&(Key,Value):void):void;
method do(t@:table[`Key,`Value], c:&(Value):void):void;
method do_allowing_updates(t@:table[`Key,`Value],
                           c:&(Value):void):void;
method keys_do(t@:table[`Key,`Value], c:&(Key):void):void;
method keys_do_allowing_updates(t@:table[`Key,`Value], c:&(Key):void):void;
```

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.

```
signature fetch(table[`Key,`Value], key:Key, if_absent:&():Value):Value;
implementation fetch(t@:table[`Key <= comparable[Key], `Value], key:Key,
                     if_absent:&():Value):Value;
method fetch(t@:table[`Key,`Value], key:Key):Value;
```

```
abstract object table_like[Key,Value];
  signature fetch(table_like[`Key,`Value], Key):Value;
  method !(t@:table_like[`Key,`Value], key:Key):Value;
extend type table_like[`Key,`Value] subtypes table_like[Key, `Value1 >= Value];
extend table[`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
```

```
method find_key(t@:table[`Key,`Value <= comparable[Value]], value:Value):Key;
method find_key(t@:table[`Key,`Value <= comparable[Value]],
                value:Value, if_absent:&():Key):Key;
method pick_any_key(t@:table[`Key,`Value]):Key;
method pick_any_key(t@:table[`Key,`Value], if_empty:&():`T):Key|T;
method includes_key(t@:table[`Key,`Value], key:Key):bool;
method keys(t@:table[`Key,`Value]):collection[Key];
method keys_set(t@:table[`Key <= comparable[Key],`Value]):set[Key];
method keys_list(t@:table[`Key,`Value]):ordered_collection[Key];
extend table[`Key, `Value <= comparable[Value]]
                                    isa comparable[table[Key,Value]];
method =(t1@:table[`Key, `Value <= comparable[Value]],
        t2@:table[Key, Value]):bool;
method collection_name(@:table[`Key,`Value]):string;
method values_print_string(t@:table[`Key,`Value]):string;
method elems_print_string(t@:table[`Key,`Value]):string;
method elems_print(t@:table[`Key,`Value]):void;
```

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.

```
abstract object i_table[Key, Value] isa table[Key, Value];
  extend type i_table[`Key, `Value] subtypes i_table[Key, `Value1 >= Value];
  method copy(t@:`T <= i_table[`Key,`Value]):T;
```

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

```
abstract object m_table[Key, Value] isa table[Key, Value];
signature store(m_table[`Key,`Value], key:Key, value:Value,
                if_absent:&():void):void;
method store(t@:m_table[`Key,`Value], key:Key, value:Value):void;
method store_no_dup(t@:m_table[`Key,`Value], key:Key, value:Value):void;
abstract object m_table_like[Key,Value] isa table_like[Key,Value];
  signature store(m_table_like[`Key,`Value], Key, Value):void;
  method set_!(t@:m_table_like[`Key,`Value], key:Key, value:Value):void;
extend m_table[`Key,`Value] isa m_table_like[Key,Value];
```

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.)

```
method fetch_or_init(t@:m_table[`Key,`Value], key:Key,
                     if_init:&():Value):Value;
method copy(t@:m_table[`Key,`Value]):m_table[Key,Value];
signature copy_empty(m_table[`Key,`Value]):m_table[Key,Value];
```

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. Method `copy_empty`, given a (mutable) table, returns an empty mutable table of the same kind.

```
abstract object removable_table[Key, Value] isa table[Key, Value];
extend type removable_table[`Key,`Value]
                        subtypes removable_table[Key, `Value1 >= Value];
signature remove_key(removable_table[`Key,`Value], key:Key,
                     if_absent:&():`Value):Value;
method remove_key(t@:removable_table[`Key,`Value], key:Key):Value;
method remove_all(t@:removable_table[`Key,`Value]):void;
method remove_keys_if(t@:removable_table[`Key,`Value],
                        pred:&(Key):bool):int;
method remove_if(t@:removable_table[`Key,`Value],
                 pred:&(Value):bool):int;
signature copy_empty(removable_table[`Key,`Value]):removable_table[Key,Value];
```

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 object m_removable_table[Key, Value]
        isa m_table[Key, Value], removable_table[Key, Value];
  signature copy(m_removable_table[`Key,`Value]):m_removable_table[Key,Value];
  signature copy_empty(m_removable_table[`Key,`Value]
                       ):m_removable_table[Key,Value];
```

`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.

4.5.1 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.cecil`:

```
let var warn_for_long_assoc_tables:bool;
template object assoc[Key <= comparable[Key], Value];
  field key(@:assoc[`Key,`Value]):Key;
  var field value(@:assoc[`Key,`Value]):Value;
  extend assoc[`Key, `Value <= comparable[Value]]
                                        isa comparable[assoc[Key,Value]];
  method =(l@:assoc[`Key, `Value <= comparable[Value]],
           r@:assoc[`Key, `Value <= comparable[Value]]):bool;
  method print_string(a@:assoc[`Key,`Value]):string;
  method new_assoc[Key <= comparable[Key], Value](k:Key, v:Value
                                                  ):assoc[Key,Value];
  method copy(a@:assoc[`Key,`Value]):assoc[Key,Value];
template object assoc_table[Key <= comparable[Key], Value]
                                        isa m_removable_table[Key,Value];
  method length(t@:assoc_table[`Key,`Value]):int;
  method is_empty(t@:assoc_table[`Key,`Value]):bool;
  method do_associations(t@:assoc_table[`Key,`Value],
                         c:&(Key,Value):void):void;
  method store(t@:assoc_table[`Key,`Value], key:Key, value:Value,
               if_absent:&():void):void;
  method store_no_dup(t@:assoc_table[`Key,`Value], key:Key, value:Value):void;
  method add_assoc(t@:assoc_table[`Key,`Value], a:assoc[Key,Value]):void;
  method remove_key(t@:assoc_table[`Key,`Value], key:Key,
                    if_absent:&():`Else):Value|Else;
  method remove_keys_if(t@:assoc_table[`Key,`Value], pred:&(Key):bool):int;
  method remove_all(t@:assoc_table[`Key,`Value]):void;
  method new_assoc_table[Key <= comparable[Key], Value]()
                                                :assoc_table[Key,Value];
  method copy_empty(t@:assoc_table[`Key,`Value]):assoc_table[Key,Value];
  method copy(t@:assoc_table[`Key,`Value]):assoc_table[Key,Value];
template object assoc_CR_table[Key <= comparable[Key], Value]
                        isa assoc_table[Key,Value];
  method elem_separator(t@:assoc_CR_table[`Key,`Value]):string;
  method new_assoc_CR_table[Key <= comparable[Key], Value]()
                           :assoc_CR_table[Key,Value];
  method copy_empty(c@:assoc_CR_table[`Key,`Value]):assoc_CR_table[Key,Value];
template object ordered_assoc_table[Key <= comparable[Key], Value]
                                        isa assoc_table[Key,Value];
  method add_assoc(t@:ordered_assoc_table[`Key,`Value],
                   a:assoc[Key,Value]):void;
  method new_ordered_assoc_table[Key <= comparable[Key], Value]()
                           :ordered_assoc_table[Key,Value];
  method copy_empty(c@:ordered_assoc_table[`Key,`Value]
                    ):ordered_assoc_table[Key,Value];
template object ordered_assoc_CR_table[Key <= comparable[Key], Value]
                        isa ordered_assoc_table[Key,Value];
  method elem_separator(t@:ordered_assoc_CR_table[`Key,`Value]):string;
  method new_ordered_assoc_CR_table[Key <= comparable[Key], Value]()
                           :ordered_assoc_CR_table[Key,Value];
  method copy_empty(c@:ordered_assoc_CR_table[`Key,`Value]
                    ):ordered_assoc_CR_table[Key,Value];
```

`assoc_table` is an implementation of tables based on a linked-list of key/value associations.

The keys must be `comparable`.

In `identity-table.cecil`:

```
template object 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;
  method =(l@:identity_assoc[`Key1,`Value1],
           r@:identity_assoc[`Key2,`Value2]):bool;
  method print_string(a@:identity_assoc[`Key,`Value]):string;
  method new_identity_assoc[Key,Value](k:Key, v:Value
                                      ):identity_assoc[Key,Value];
  method copy(a@:identity_assoc[`Key,`Value]):identity_assoc[Key,Value];
template object identity_assoc_table[Key,Value]
                isa m_removable_table[Key,Value];
  method length(t@:identity_assoc_table[`Key,`Value]):int;
  method is_empty(t@:identity_assoc_table[`Key,`Value]):bool;
  method do_associations(t@:identity_assoc_table[`Key,`Value],
                         c:&(Key,Value):void):void;
  method fetch(t@:identity_assoc_table[`Key,`Value], key:Key,
               if_absent:&():Value):Value;
  method store(t@:identity_assoc_table[`Key,`Value], key:Key, value:Value,
               if_absent:&():void):void;
  method remove_key(t@:identity_assoc_table[`Key,`Value], key:Key,
                    if_absent:&():`Else):Value|Else;
  method remove_keys_if(t@:identity_assoc_table[`Key,`Value],
                        pred:&(Key):bool):int;
  method remove_all(t@:identity_assoc_table[`Key,`Value]):void;
  method new_identity_assoc_table[Key,Value]():identity_assoc_table[Key,Value];
  method copy_empty(t@:identity_assoc_table[`Key,`Value]
                   ):identity_assoc_table[Key,Value];
  method copy(t@:identity_assoc_table[`Key,`Value]
             ):identity_assoc_table[Key,Value];
template object identity_assoc_CR_table[Key,Value]
                        isa identity_assoc_table[Key,Value];
  method elem_separator(t@:identity_assoc_CR_table[`Key,`Value]):string;
  method new_identity_assoc_CR_table[Key,Value]()
                           :identity_assoc_CR_table[Key,Value];
  method copy_empty(c@:identity_assoc_CR_table[`Key,`Value]
                   ):identity_assoc_CR_table[Key,Value];
```

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

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.cecil`:

```
concrete object empty_hash_bucket;
concrete object removed_hash_bucket;
let use_linear_probing:bool;
```

```
method buckets_in_probing_order_do(v@:vector[`T], start_idx@:int,
                                   cl:&(int,T):void):void;
method buckets_in_linear_probing_order_do(v@:vector[`T], start_idx_oop@:int,
                                          cl:&(int,T):void
                                          ):void;
method next_probe(v@:vector[`T], idx:int):int;
method previous_probe(v@:vector[`T], idx:int):int;
method buckets_in_quadratic_probing_order_do(v@:vector[`T], start_idx@:int,
                                             cl:&(int,T):void
                                             ):void;
let good_table_sizes:vector[int];
let var default_open_table_size:int;
abstract object open_table[Key];
method remove_bucket(t@:open_table[`Key], idx:int):void;
method remove_all(t@:open_table[`Key]):void;
method probe_histogram(t@:open_table[`Key <= hashable[Key]]):histogram[int];

template object hash_table[Key <= hashable[Key], Value]
                        isa m_removable_table[Key,Value], open_table[Key];
method do_associations(t@:hash_table[`Key,`Value], c:&(Key,Value):void):void;
method do_associations_allowing_updates(t@:hash_table[`Key,`Value],
                                        c:&(Key,Value):void):void;
method fetch(t@:hash_table[`Key,`Value], key:Key, if_absent:&():Value):Value;
method store(t@:hash_table[`Key,`Value], key:Key, value:Value,
             if_absent:&():void):void;
method remove_key(t@:hash_table[`Key,`Value], key:Key,
                  if_absent:&():`Else):Value|Else;
method remove_keys_if(t@:hash_table[`Key,`Value], pred:&(Key):bool):int;
method remove_all(t@:hash_table[`Key,`Value]):void;
method check_correctness(t@:hash_table[`Key,`Value]):void;
method new_hash_table[Key <= hashable[Key], Value]():hash_table[Key,Value];
method new_hash_table[Key <= hashable[Key], Value](size:int)
                                                    :hash_table[Key,Value];
method copy_empty(t@:hash_table[`Key,`Value]):hash_table[Key,Value];
method copy(t@:hash_table[`Key,`Value]):hash_table[Key,Value];
template object hash_CR_table[Key <= hashable[Key], Value]
                                                isa hash_table[Key,Value];
method elem_separator(t@:hash_CR_table[`Key,`Value]):string;
method new_hash_CR_table[Key <= hashable[Key], Value]()
                                               :hash_CR_table[Key,Value];
method new_hash_CR_table[Key <= hashable[Key], Value](size:int)
                                               :hash_CR_table[Key,Value];
method copy_empty(c@:hash_CR_table[`Key,`Value]):hash_CR_table[Key,Value];
method copy(t@:hash_CR_table[`Key,`Value]):hash_CR_table[Key,Value];
```

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

In `chained-hash-table.cecil`:

```
template object chained_hash_table[Key <= hashable[Key], Value]
                                       isa m_removable_table[Key,Value];
method do_associations(t@:chained_hash_table[`Key,`Value],
                       c:&(Key,Value):void):void;
```

```
method do_associations_allowing_updates(t@:chained_hash_table[`Key,`Value],
                                        c:&(Key,Value):void):void;
method fetch(t@:chained_hash_table[`Key,`Value], key:Key,
             if_absent:&():`Else):Value|Else;
method store(t@:chained_hash_table[`Key,`Value], key:Key, value:Value,
             if_absent:&():void):void;
method remove_key(t@:chained_hash_table[`Key,`Value], key:Key,
                  if_absent:&():`Else):Value|Else;
method remove_keys_if(t@:chained_hash_table[`Key,`Value],
                        pred:&(Key):bool):int;
method remove_all(t@:chained_hash_table[`Key,`Value]):void;
let var default_chained_hash_table_size:int;
method new_chained_hash_table[Key <= hashable[Key], Value]()
     :chained_hash_table[Key,Value];
method new_chained_hash_table[Key <= hashable[Key], Value](size:int)
     :chained_hash_table[Key,Value];
method copy_empty(t@:chained_hash_table[`Key,`Value]
                  ):chained_hash_table[Key,Value];
method copy(t@:chained_hash_table[`Key,`Value]):chained_hash_table[Key,Value];
method probe_histogram(t@:chained_hash_table[`Key,`Value]):histogram[int];
template object chained_hash_CR_table[Key <= hashable[Key], Value]
        isa chained_hash_table[Key,Value];
method elem_separator(t@:chained_hash_CR_table[`Key,`Value]):string;
method new_chained_hash_CR_table[Key <= hashable[Key], Value]()
     :chained_hash_CR_table[Key,Value];
method new_chained_hash_CR_table[Key <= hashable[Key], Value](size:int)
                                            :chained_hash_CR_table[Key,Value];
method copy_empty(c@:chained_hash_CR_table[`Key,`Value]
                  ):chained_hash_CR_table[Key,Value];
```

chained_hash_table is an implementation of tables using a closed hashing algorithm.

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

In `indexed.cecil`:

```
abstract object indexed[T] isa sequence[T], table[int,T];
extend type indexed[`T] subtypes indexed[`S >= T];
method !(t@:indexed[`T], key:int): T;
method fetch(t@:indexed[`T], key:int): T;
method do(a@:indexed[`T], c:&(T):void):void;
method do(a1@:indexed[`T1], a2@:indexed[`T2], c:&(T1,T2):void):void;
method do(a1@:indexed[`T1], a2@:indexed[`T2], a3@:indexed[`T3],
          c:&(T1,T2,T3):void):void;
method do(a1@:indexed[`T1], a2@:indexed[`T2], a3@:indexed[`T3],
          a4@:indexed[`T4], c:&(T1,T2,T3,T4):void):void;
method reverse_do(a@:indexed[`T], c:&(T):void):void;
method reverse_do(a1@:indexed[`T1], a2@:indexed[`T2],
                  c:&(T1,T2):void):void;
method reverse_do(a1@:indexed[`T1], a2@:indexed[`T2], a3@:indexed[`T3],
          c:&(T1,T2,T3):void):void;
method reverse_do(a1@:indexed[`T1], a2@:indexed[`T2], a3@:indexed[`T3],
```

```
                  a4@:indexed[`T4], c:&(T1,T2,T3,T4):void):void;
method do_associations(a@:indexed[`T], c:&(int,T):void):void;
method do_associations_allowing_updates(a@:indexed[`T], c:&(int,T):void):void;
method includes_key(a@:indexed[`T], key:int):bool;
method keys(t@:indexed[`T]):interval;
method ||(s1@:indexed[`T], s2@:indexed[`T]):indexed[T];
```

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.

Currently, none of the `_allowing_updates` iterators work for indexed collections.

```
method includes_index(a@:indexed[`T], i:int):bool;
method find_index(a@:indexed[`T <= comparable[T]], value:T):int;
method find_index(a@:indexed[`T <= comparable[T]], value:T,
                  if_absent:&():int):int;
method elems_print_string(c@:indexed[`T]):string;
method elems_print(c@:indexed[`T]):void;
method =(c1@:indexed[`T <= comparable[T]], c2@:indexed[T]):bool;
method <(c1@:indexed[`T <= ordered[T]], c2@:indexed[T]):bool;
method <=(c1@:indexed[`T <= ordered[T]], c2@:indexed[T]):bool;
signature copy(indexed[`T]):indexed[T];
```

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

```
abstract object i_indexed[T] isa indexed[T], i_table[int,T];
extend type i_indexed[`T] subtypes i_indexed[`S >= T];
abstract object m_indexed[T] isa indexed[T], m_table[int,T];
  method do_associations_allowing_updates(a@:m_indexed[`T],
                                          c:&(int,T):void):void;
method as_m_indexed(t@:m_indexed[`T]):m_indexed[T];
signature copy(m_indexed[`T]):m_indexed[T];
```

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`.

```
method set_first (a@:m_indexed[`T], x:T):void;
method set_second(a@:m_indexed[`T], x:T):void;
method set_third (a@:m_indexed[`T], x:T):void;
method set_fourth(a@:m_indexed[`T], x:T):void;
method set_last  (a@:m_indexed[`T], x:T):void;
```

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`.

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

The swap method exchanges the indices of two collection elements.

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

```
method pos(s1:indexed[`T <= comparable[T]], s2@:indexed[T]):int;
method pos(s1@:indexed[`T <= comparable[T]], s2@:indexed[T],
           if_fail:&():int):int;
method has_subsequence(s1@:indexed[`T <= comparable[T]], s2@:indexed[T]):bool;
method count_subsequences(s1@:indexed[`T <= comparable[T]],
                          s2@:indexed[T]):int;
```

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!]

4.6.1 Implementations

Several concrete implementations of indexed collections exist:

- `i_vector`s are primitive fixed-length immutable indexed collections, item `m_vector`s are primitive fixed-length mutable indexed collections,
- `array`s are extensible mutable indexed collections,
- `interval`s are immutable indexed collections representing a finite arithmetic sequence of integers,
- `string`s are indexed collections of characters, with mutable and immutable varieties,
- `bit_vector`s are fixed-length mutable indexed collections of 0/1 values,
- `word_vector`s are fixed-length mutable indexed collections of single-word short integers, and
- `float_vector`s are fixed-length mutable indexed collections of floating-point values (short integers).

In `vector.cecil`:

```
abstract object vector[`T] isa indexed[T];
extend type vector[`T] subtypes vector[`S >= T];
method as_vector(v@:vector[`T]):vector[T];
method length(v@:vector[`T]):int;
method fetch(v@:vector[`T], index:int):T;
method fetch(v@:vector[`T], index:int, if_absent:&():T):T;
extend i_vector[`T] isa vector[T], i_indexed[T];
extend type i_vector[`T] subtypes i_vector[`S >= T];
method new_i_vector[T](size@:int, filler:T):i_vector[T];
method new_i_vector(size@:int, filler:dynamic):i_vector[dynamic];
method new_i_vector_init[T](size@:int, cl:&(int):T):i_vector[T];
method new_i_vector_init_from[T2](c@:ordered_collection[`T1],
```

```
                                  cl:&(T1):T2):i_vector[T2];
method new_i_vector_init_from[T2](c@:indexed[`T1],
                                  cl:&(T1):T2):i_vector[T2];
method new_i_vector_init_from[T2](c@:vector[`T1],
                                    cl:&(T1):T2):i_vector[T2];
method as_i_vector(v@:i_vector[`T]):i_vector[T];
method collection_name(@:i_vector[`T]):string;
extend m_vector[`T] isa vector[T], m_indexed[T];
method store(v@:m_vector[`T], index:int, x:T, if_absent:&():void):void;
method new_m_vector[T](size@:int):m_vector[T];
method new_m_vector[T](size@:int, filler:T):m_vector[T];
method new_m_vector(size@:int):m_vector[dynamic];
method new_m_vector(size@:int, filler:dynamic):m_vector[dynamic];
method new_m_vector_init[T](size@:int, cl:&(int):T):m_vector[T];
method new_m_vector_init_from[T2](c@:ordered_collection[`T1],
                                    cl:&(T1):T2):m_vector[T2];
method new_m_vector_init_from[T2](c@:indexed[`T1],
                                    cl:&(T1):T2):m_vector[T2];
method new_m_vector_init_from[T2](c@:vector[`T1],
                                    cl:&(T1):T2):m_vector[T2];
method as_m_vector(v@:m_vector[`T]):m_vector[T];
method copy(v@:m_vector[`T]):m_vector[T];
method collection_name(@:m_vector[`T]):string;
```

`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 Cecil 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.

In `float-vector.cecil`:

```
abstract object float_vector isa indexed[float];
method as_float_vector(v@:float_vector):float_vector;
method length(v@:float_vector):int;
method fetch(v@:float_vector, index:int, if_absent:&():float):float;
extend i_float_vector isa float_vector, i_indexed[float];
method new_i_float_vector(size@:int, filler_oop@:float):i_float_vector;
method new_i_float_vector_init(size@:int, cl:&(int):float):i_float_vector;
method new_i_float_vector_init_from(c@:indexed[`T],
                                  cl:&(T):float):i_float_vector;
method new_i_float_vector_init_from(c@:ordered_collection[`T],
                                  cl:&(T):float):i_float_vector;
method as_i_float_vector(v@:i_float_vector):i_float_vector;
method copy(v@:i_float_vector):i_float_vector;
method collection_name(@:i_float_vector):string;
```

```
extend m_float_vector isa float_vector, m_indexed[float];
method store(v@:m_float_vector, index:int, x:float, if_absent:&():void):void;
method new_m_float_vector(size@:int):m_float_vector;
method new_m_float_vector(size@:int, filler_oop@:float):m_float_vector;
method new_m_float_vector_init(size@:int, cl:&(int):float):m_float_vector;
method new_m_float_vector_init_from(c@:indexed[`T], cl:&(T):float
                                   ):m_float_vector;
method new_m_float_vector_init_from(c@:ordered_collection[`T],
                                    cl:&(T):float):m_float_vector;
method as_m_float_vector(v@:m_float_vector):m_float_vector;
method copy(v@:m_float_vector):m_float_vector;
method collection_name(@:m_float_vector):string;
```

In word-vector.cecil:

```
abstract object word_vector isa indexed[int];
method as_word_vector(v@:word_vector):word_vector;
let num_word_bits:int;
method length(v@:word_vector):int;
method fetch(v@:word_vector, index:int, if_absent:&():int):int;
extend i_word_vector isa word_vector, i_indexed[int];
method new_i_word_vector(size@:int, filler_oop@:int):i_word_vector;
method new_i_word_vector_init(size@:int, cl:&(int):int):i_word_vector;
method new_i_word_vector_init_from(c@:indexed[`T],
                               cl:&(T):int):i_word_vector;
method new_i_word_vector_init_from(c@:ordered_collection[`T],
                               cl:&(T):int):i_word_vector;
method as_i_word_vector(v@:i_word_vector):i_word_vector;
method copy(v@:i_word_vector):i_word_vector;
method collection_name(@:i_word_vector):string;
extend m_word_vector isa word_vector, m_indexed[int];
method store(v@:m_word_vector, index:int, x:int, if_absent:&():void):void;
method new_m_word_vector(size@:int):m_word_vector;
method new_m_word_vector(size@:int, filler_oop@:int):m_word_vector;
method new_m_word_vector_init(size@:int, cl:&(int):int):m_word_vector;
method new_m_word_vector_init_from(c@:indexed[`T], cl:&(T):int
                                   ):m_word_vector;
method new_m_word_vector_init_from(c@:ordered_collection[`T],
                                    cl:&(T):int):m_word_vector;
method as_m_word_vector(v@:m_word_vector):m_word_vector;
method copy(v@:m_word_vector):m_word_vector;
method collection_name(@:m_word_vector):string;
```

In array.cecil:

```
template object array[T] isa m_indexed[T], extensible_sequence[T];
extend array[`T <= comparable[T]] isa removable_collection[T];
let default_array_size:int;
method new_array[T]():array[T];
method new_array[T](size:int):array[T];
method new_array[T](size:int, filler:T):array[T];
method new_array_init[T](size:int, cl:&(int):T):array[T];
method new_array_init_from[T2](c@:indexed[`T1], cl:&(T1):T2):array[T2];
```

```
method new_array_init_from[T2](c@:ordered_collection[`T1],
                               cl:&(T1):T2):array[T2];
method as_array(c@:ordered_collection[`T]):array[T];
method as_array[T](c@:ordered_collection[T]):array[T];
method new_array():array[dynamic];
method new_array(size:int):array[dynamic];
method new_array(size:int, filler:dynamic):array[dynamic];
method fetch(a@:array[`T], index:int, if_absent:&():T):T;
method fetch(a@:array[`T], index:int):T;
method store(a@:array[`T], index:int, x:T, if_absent:&():void):void;
method add_first(a@:array[`T], x:T):void;
method add_last(a@:array[`T], x:T):void;
method ||(a1@:array[`T], a2@:array[T]):array[T];
method remove_key(a@:array[`T], index:int):T;
method remove_key(a@:array[`T], index:int, if_absent:&():`S):T|S;
method remove_first(a@:array[`T], if_empty:&():`S):T|S;
method remove_last(a@:array[`T], if_empty:&():`S):T|S;
method remove_last_N(a@:array[`T], n:int):void;
method remove(a@:array[`T <= comparable[T]], x:T, if_absent:&():void):void;
method remove_all(a@:array[`T]):void;
method copy_empty(a@:array[`T]):array[T];
method copy(a@:array[`T]):array[T];
method collection_name(@:array[`T]):string;
```

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.

In `interval.cecil`:

```
template object interval isa i_indexed[int];
 field step(@:interval):int;
method in_range(s@:interval, i:int):bool;
method length(s@:interval):int;
method do(s@:interval, c:&(int):void):void;
method do_associations(s@:interval, c:&(int,int):void):void;
method fetch(s@:interval, k:int, if_absent:&():int):int;
method includes(s@:interval, i:int):bool;
method print_string(s@:interval):string;
method collection_name(@:interval):string;
method new_interval(start:int, stop:int):interval;
method new_interval(start:int, stop:int, step:int):interval;
```

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.

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

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

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

can be expressed in Cecil 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(...).)

In `bit-vector.cecil`:

```
template object bit_vector isa m_indexed[int];
  method length(t@:bit_vector):int;
method new_bit_vector(sz:int):bit_vector;
method copy(t@:bit_vector):bit_vector;
method resize(t@:bit_vector, new_bits:int):void;
method =(l@:bit_vector, r@:bit_vector):bool;
method =_or_zero(l@:bit_vector, r@:bit_vector):bool;
```

Bit vectors are 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.

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

```
method is_disjoint(l@:bit_vector, r@:bit_vector):bool;
method fetch(v@:bit_vector, bit:int, if_absent:&():int):int;
method store(v@:bit_vector, bit:int, value:int, if_absent:&():void):void;
method set_all_bits(v@:bit_vector):void;
method clear_all_bits(v@:bit_vector):void;
method is_all_zeros(v@:bit_vector):bool;
method is_all_ones(v@:bit_vector):bool;
```

A bit vector can be mutated to be all zeros (`clear_all_bits`) or all ones (`set_all_bits`) or tested for those conditions. The `is_disjoint` method returns true if its arguments share no set bits; is_disjoint(a,b) is equivalent to `is_all_zeros(bit_and(a,b))`.

```
method do_ones(v@:bit_vector, cl:&(int):void):void;
method collection_name(@:bit_vector):string;
method elem_separator(@:bit_vector):string;
```

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.

4.6.2 Strings

In `string.cecil`:

```
abstract object string isa indexed[char];
method copy_mutable(s@:string):m_string;
```

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.

```
method ||(s1@:string, s2@:string):string;
method ||(s1@:string, c@:char):string;
method ||(c@:char, s2@:string):string;
method to_upper_case(s@:string):string;
method to_lower_case(s@:string):string;
```

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.)

```
method copy_from(s@:string, start:int):string;
method copy_from(s@:string, start:int, up_to:int):string;
method has_prefix(s@:string, prefix@:string):bool;
method has_suffix(s@:string, suffix@:string):bool;
method remove_prefix(s@:string, prefix@:string):string;
method remove_suffix(s@:string, suffix@:string):string;
```

The `copy_from` method copies a portion of a string from a start index up to (but not including) a stop index (or the end of the string, if the stop index is not specified). The `has_{prefix,suffix}` functions test whether a string starts with or ends with a particular string; the `remove_{prefix,suffix}` functions return a new string with the specified prefix or suffix removed, if present, or the original string otherwise.

```
method pad(s@:string, len:int):string;
method pad_right(s@:string, len:int):string;
method pad_right(s@:string, len:int, padding:char):string;
method pad_left(s@:string, len:int):string;
method pad_left(s@:string, len:int, padding:char):string;
method  open_brace(@:string):string;
method close_brace(@:string):string;
method collection_name(@:string):string;
method elem_print_string(@:string, char:char, :bool):string;
method print(s@:string):void;
```

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.

```
abstract object i_string isa string, i_indexed[char];
abstract object m_string isa string, m_indexed[char];
  method copy(s@:m_string):m_string;
  method write_into_string_at_pos(s1:string, s2:m_string, pos:int):void;
```

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

```
abstract object vstring isa string;
method length(s@:vstring):int;
method fetch(s@:vstring, index:int, if_absent:&():char):char;
method as_vstring(s@:vstring):vstring;
method print(s@:vstring):void;
extend i_vstring isa i_string, vstring;
method new_i_vstring(size@:int):i_vstring;
method new_i_vstring(size@:int, filler_oop@:char):i_vstring;
method new_i_vstring_init(size@:int, cl:&(int):char):i_vstring;
method new_i_vstring_init_from(c@:ordered_collection[`T],
                               cl:&(T):char):i_vstring;
method new_i_vstring_init_from(c@:indexed[`T], cl:&(T):char
                               ):i_vstring;
extend m_vstring isa m_string, vstring;
method new_m_vstring(size@:int):m_vstring;
method new_m_vstring_no_init(size@:int):m_vstring;
method new_m_vstring(size@:int, filler_oop@:char):m_vstring;
method new_m_vstring_init(size@:int, cl:&(int):char):m_vstring;
method new_m_vstring_init_from(c@:ordered_collection[`T],
                               cl:&(T):char):m_vstring;
method new_m_vstring_init_from(c@:indexed[`T], cl:&(T):char
                               ):m_vstring;
method store(s@:m_vstring, index:int, c:char, if_absent:&():void):void;
method as_m_vstring(s@:m_vstring):m_vstring;
```

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

4.7 Lists

In list.cecil:

```
abstract object list[T] isa sequence[T];
extend type list[`T] subtypes list[`S >= T];
signature first(list[`T]):T;
signature rest(list[`T]):list[T];
method collection_name(@:list[`T]):string;
predicate empty_list[T] isa list[T], empty_collection[T];
predicate non_empty_list[T] isa list[T], non_empty_collection[T];
implementation first(@:empty_list[`T]):none;
implementation rest(@:empty_list[`T]):none;
```

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.

```
abstract object simple_list[T] isa list[T],
                                   functionally_extensible_collection[T];
var field signature first(simple_list[`T]):T;
var field signature rest(simple_list[`T]):simple_list[T];
signature copy(simple_list[`T]):simple_list[T];
method add_functional(l@:simple_list[`T], x:T):simple_list[T];
method do(c1@:simple_list[`T1], c2@:simple_list[`T2],
               c:&(T1, T2):void):void;
method do(c1@:simple_list[`T1], c2@:simple_list[`T2],
                 c3@:simple_list[`T3], c:&(T1, T2, T3):void):void;
concrete representation nil[T] isa simple_list[T];
method first(@:nil[`T]):none;
method rest(@:nil[`T]):none;
method length(@:nil[`T]):int;
method is_empty(@:nil[`T]):bool;
method do(@:nil[`T], :&(T):void):void;
method reverse_do(@:nil[`T], :&(T):void):void;
method copy(n@:nil[`T]):simple_list[T];
method set_first(@:nil[`T], :T):none;
method set_rest(@:nil[`T], :simple_list[T]):none;
template representation cons[T] isa simple_list[T];
var field first(@:cons[`T]):T;
var field rest(@:cons[`T]):simple_list[T];
method length(c@:cons[`T]):int;
method is_empty(@:cons[`T]):bool;
method do(c@:cons[`T], closure:&(T):void):void;
method reverse_do(c@:cons[`T], closure:&(T):void):void;
method do(c1@:cons[`T1], c2@:cons[`T2], c:&(T1,T2):void):void;
method do(c1@:cons[`T1], c2@:cons[`T2], c3@:cons[`T3],
                 c:&(T1,T2,T3):void):void;
method copy(c@:cons[`T]):simple_list[T];
method cons(hd:T, tl@:simple_list[`T]):cons[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.

```
template object m_list[T] isa list[T], extensible_sequence[T];
extend m_list[`T <= comparable[T]] isa removable_collection[T];
method reverse_do(l@:m_list[`T], c:&(T):void):void;
method do(l@:m_list[`T], c:&(T):void):void;
method do(l1@:m_list[`T1], l2@:m_list[`T2], c:&(T1,T2):void):void;
method do(l1@:m_list[`T1], l2@:m_list[`T2], l3@:m_list[`T3],
                 c:&(T1,T2,T3):void):void;
method first(l@:m_list[`T]):T;
```

```
method second(l@:m_list[`T]):T;
method rest(l@:m_list[`T]):m_list[T];
method add(l@:m_list[`T], x:T):void;
method add_first(l@:m_list[`T], x:T):void;
method add_last(l@:m_list[`T], x:T):void;
method remove(l@:m_list[`T <= comparable[T]], x:T, if_absent:&():void):void;
method remove_and_return_one(l@:m_list[`T], test:&(T):bool,
                             if_absent:&():`Else):T|Else;
method remove_if(l@:m_list[`T], test:&(T):bool):int;
method remove_first(l@:m_list[`T], if_empty:&():`S):T|S;
method remove_last(l@:m_list[`T], if_empty:&():`S):T|S;
method splice_onto_end(l1@:m_list[`T], l2:m_list[T]):m_list[T];
method remove_all(l@:m_list[`T]):void;
method new_m_list[T]():m_list[T];
method copy_empty(l@:m_list[`T]):m_list[T];
method copy(l@m_list[`T]:`L <= m_list[`T]):LC
                                where signature copy_empty(L):`LC;
```

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.

4.8 Sorted collections

In `sorted.cecil`:

```
abstract object sorted_collection[T] isa ordered_collection[T];
extend type sorted_collection[`T] subtypes sorted_collection[`S >= T];
method view_stream(c@:sorted_collection[`T]):stream[T];
abstract object m_sorted_collection[T] isa sorted_collection[T],
                                            extensible_ordered[T];
```

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.

```
abstract object simple_binary_tree[T <= ordered[T]] isa sorted_collection[T];
```

```
signature add(simple_binary_tree[`T], x:T):simple_binary_tree[T];
method collection_name(@:simple_binary_tree[`T]):string;
signature copy(simple_binary_tree[`T]):simple_binary_tree[T];
concrete representation empty_tree[T <= ordered[T]] isa simple_binary_tree[T];
method length(t@:empty_tree[`T]):int;
method do(t@:empty_tree[`T], c:&(T):void):void;
method reverse_do(t@:empty_tree[`T], c:&(T):void):void;
method add(t@:empty_tree[`T], x:T):simple_binary_tree[T];
method copy(t@:empty_tree[`T]):simple_binary_tree[T];
template representation tree_node[T <= ordered[T]] isa simple_binary_tree[T];
method length(t@:tree_node[`T]):int;
method is_empty(t@:tree_node[`T]):bool;
method do(t@:tree_node[`T], c:&(T):void):void;
method reverse_do(t@:tree_node[`T], c:&(T):void):void;
method add(t@:tree_node[`T], x:T):simple_binary_tree[T];
method copy(t@:tree_node[`T]):simple_binary_tree[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`.

```
template object m_binary_tree[T <= ordered[T]] isa m_sorted_collection[T];
method length(t@:m_binary_tree[`T]):int;
method is_empty(t@:m_binary_tree[`T]):bool;
method do(t@:m_binary_tree[`T], c:&(T):void):void;
method reverse_do(t@:m_binary_tree[`T], c:&(T):void):void;
method add(t@:m_binary_tree[`T], x:T):void;
method collection_name(@:m_binary_tree[`T]):string;
method new_sorted_collection[T <= ordered[T]]():m_binary_tree[T];
method copy(t@:m_binary_tree[`T]):m_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.

In `skiplist.cecil`:

```
template object skip_list[T <= comparable[T]] isa m_sorted_collection[T],
                                                  removable_collection[T];
  method less_than(t@:skip_list[`T], e1:T, e2:T):bool;
  method less_than(t@:skip_list[`T],
                   e1:T, e2@:skip_list_nil_value):bool;
  method less_than(t@:skip_list[`T],
                   e1@:skip_list_nil_value, e2:T):bool;
  method less_than(t@:skip_list[`T],
                   e1@:skip_list_nil_value,e2@:skip_list_nil_value):bool;
  method new_skip_list[T <= partially_ordered[T]]():skip_list[T];
  method new_predicate_skip_list[T <= comparable[T]](cmp:&(T,T):bool
                                                     ):skip_list[T];
  method add(l@:skip_list[`T], new_value:T):void;
```

```
  method includes(l@:skip_list[`T], elem:T):bool;
  method do(l@:skip_list[`T], c:&(T):void):void;
  method remove(l@:skip_list[`T], value:T, if_absent:&():void):void;
  method length(l@:skip_list[`T]):int;
  method is_empty(l@:skip_list[`T]):bool;
  method first(l@:skip_list[`T]):T;
  method last(l@:skip_list[`T]):T;
  method remove_first(l@:skip_list[`T], if_empty:&():`S):T|S;
  method remove_last(l@:skip_list[`T], if_empty:&():`S):T|S;
  method collection_name(@:skip_list[`T]):string;
  method copy(l@:skip_list[`T]):skip_list[T];
  method copy_empty(l@:skip_list[`T]):skip_list[T];
template object skip_list_node[T];
  field value(@:skip_list_node[`T]):T;
  field forward(@:skip_list_node[`T]):array[skip_list_node[T]];
  method new_skip_list_node[`T](level:int, new_value:T):skip_list_node[T];
  method is_nil(sln@:skip_list_node[`T]):bool;
  method print_string(sln@:skip_list_node[`T]):string;
  method copy(sln@:skip_list_node[`T]):skip_list_node[T];
concrete representation skip_list_nil[T] subtypes skip_list_node[T];
  method value(@:skip_list_nil[`T]):T;
  method is_nil(@:skip_list_nil[`T]):bool;
  method print_string(@:skip_list_nil[`T]):string;
  method copy(l@:skip_list_nil[`T]):skip_list_nil[T];
concrete representation skip_list_nil_value isa ordered[`T];
  method = (x1@:skip_list_nil_value, x2@:skip_list_nil_value):bool;
  method = (x1@:skip_list_nil_value, x2@:comparable[`T]):bool;
  method = (x1@:comparable[`T], x2@:skip_list_nil_value):bool;
  method < (x1@:skip_list_nil_value, x2@:skip_list_nil_value):bool;
  method < (x1@:skip_list_nil_value, x2@:ordered[`T]):bool;
  method < (x1@:ordered[`T], x2@:skip_list_nil_value):bool;
concrete representation rand_sl_level_stream;
  field rs(@:rand_sl_level_stream):random_stream;
  method next(r@:rand_sl_level_stream, current_level:int):int;
```

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.

4.9 stack, queue

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

In `stack.cecil`:

```
template object stack[T];
  method collection_name(@:stack[`T]):string;
  method push(s@:stack[`T], x:T):void;
  method pop(s@:stack[`T]):T;
  method top(s@:stack[`T]):T;
  method copy(s@:stack[`T]):stack[T];
  method new_stack[T]():stack[T];
```

Stacks are implemented as special interfaces to mutable lists.

In `queue.cecil`:

```
template object queue[T];
  method collection_name(t@:queue[`T]):string;
  method enqueue(t@:queue[`T], x:T):void;
  method dequeue(t@:queue[`T]):T;
  method dequeue(t@:queue[`T], if_empty:&():T):T;
  method copy(q@:queue[`T]):queue[T];
  method new_queue[T]():queue[T];
```

Queues are implemented as special interfaces to mutable lists.

4.10 Keyed sets

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

In `keyed-set.cecil`:

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.

```
abstract object keyed_comparable[Key <= comparable[Key]];
  signature key(keyed_comparable[`Key]):Key;
  method has_key(k1@:keyed_comparable[`Key], k2:Key):bool;
  method is_same_key(k1@:keyed_comparable[`Key],
                     k2@:keyed_comparable[`Key]):bool;
```

Keyed set elements must be subtypes of `keyed_comparable`:

```
abstract object keyed_set[Key <= comparable[Key],
                          Value <= keyed_comparable[Key]]
                isa table[Key,Value];
  extend type keyed_set[`Key,`Value] subtypes keyed_set[Key, `Value1 >= Value];
  method do_associations(t@:keyed_set[`Key,`Value], c:&(Key,Value):void):void;
  method do_associations_allowing_updates(t@:keyed_set[`Key,`Value],
                                          c:&(Key,Value):void):void;
  method fetch(t@:keyed_set[`Key,`Value], key:Key, if_absent:&():Value):Value;
  method match(t@:keyed_set[`Key,`Value], v:Value, if_absent:&():Value):Value;
  method match(t@:keyed_set[`Key,`Value], v:Value):Value;
  method find_key(t@:keyed_set[`Key,`Value <= comparable[Value]], value:Value,
                  if_absent:&():Key):Key;
```

Keyed sets also support `removable_` and `extensible_collection` operations such as `remove`, `add`, `add_all`, `add_no_dup`. 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).

```
  extend keyed_set[`Key, `Value <= comparable[Value]]
        isa unordered_collection[Value], comparable[keyed_set[Key,Value]];
  method =(t1@:keyed_set[`Key, `Value <= comparable[Value]],
           t2@:keyed_set[Key, Value]):bool;
  extend keyed_set[`Key, `Value <= hashable[Value]]
```

```
                                      isa hashable[keyed_set[Key,Value]];
  method hash(t@:keyed_set[`Key,`Value <= hashable[Value]], range:int):int;
  method elems_print_string(t@:keyed_set[`Key,`Value]):string;
  method elems_print(t@:keyed_set[`Key,`Value]):void;
```

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).

```
abstract object i_keyed_set[Key <= comparable[Key],
                            Value <= keyed_comparable[Key]]
                    isa keyed_set[Key,Value], i_table[Key,Value];
  extend i_keyed_set[`Key, `Value <= comparable[Value]]
        isa i_unordered_collection[Value];
  extend type i_keyed_set[`Key,`Value]
        subtypes i_keyed_set[Key, `Value1 >= Value];
  method copy(t@:i_keyed_set[`Key,`Value]):i_keyed_set[Key,Value];
abstract object m_keyed_set[Key <= comparable[Key],
                            Value <= keyed_comparable[Key]]
                isa keyed_set[Key,Value], m_removable_table[Key,Value],
                    extensible_collection[Value];
  extend m_keyed_set[`Key, `Value <= comparable[Value]]
        isa m_unordered_collection[Value], removable_collection[Value];
  method fetch_or_init(t@:m_keyed_set[`Key,`Value], key:Key,
                       if_init:&():Value):Value;
  method store(t@:m_keyed_set[`Key,`Value], key:Key, value:Value,
               if_absent:&():void):void;
```

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)`.

```
  signature remove_match(m_keyed_set[`Key,`Value], value:Value,
                         if_absent:&():`Else):Value|Else;
  method remove_match(t@:m_keyed_set[`Key,`Value], value:Value):Value;
  signature copy_empty(m_keyed_set[`Key,`Value]):m_keyed_set[Key,Value];
  method copy(t@:m_keyed_set[`Key,`Value]):m_keyed_set[Key,Value];
  signature add_nonmember(m_keyed_set[`Key,`Value], Value):void;
```

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

```
template object list_keyed_set[Key <= comparable[Key],
                               Value <= keyed_comparable[Key]]
                                        isa m_keyed_set[Key,Value];
  method collection_name(@:list_keyed_set[`Key,`Value]):string;
  method length(t@:list_keyed_set[`Key,`Value]):int;
  method is_empty(t@:list_keyed_set[`Key,`Value]):bool;
  method do(t@:list_keyed_set[`Key,`Value], c:&(Value):void):void;
  method add(m@:list_keyed_set[`Key,`Value], x:Value):void;
  method add_nonmember(m@:list_keyed_set[`Key,`Value], x:Value):void;
```

```
  method remove_key(t@:list_keyed_set[`Key,`Value], key:Key,
                    if_absent:&():`Else):Value|Else;
  method remove_match(t@:list_keyed_set[`Key,`Value], x:Value,
                         if_absent:&():`Else):Value|Else;
  method remove(t@:list_keyed_set[`Key,`Value <= comparable[Value]], x:Value,
                if_absent:&():void):void;
  method remove_any(t@:list_keyed_set[`Key,`Value],
                    if_empty:&():`Else):Value|Else;
  method remove_all(t@:list_keyed_set[`Key,`Value]):void;
  method new_list_keyed_set[Key <= comparable[Key],
                            Value <= keyed_comparable[Key]](
        ):list_keyed_set[Key,Value];
  method copy_empty(c@:list_keyed_set[`Key,`Value]):list_keyed_set[Key,Value];
```

A linked-list based implementation of `m_keyed_sets`.

```
abstract object keyed_hashable[Key <= hashable[Key]] isa keyed_comparable[Key];
  method hash_key(k@:keyed_hashable[`Key], range:int):int;
template object chained_hash_keyed_set[Key <= hashable[Key],
                                       Value <= keyed_hashable[Key]]
                                            isa m_keyed_set[Key,Value];
  method collection_name(@:chained_hash_keyed_set[`Key,`Value]):string;
  method do(t@:chained_hash_keyed_set[`Key,`Value], c:&(Value):void):void;
  method do_allowing_updates(t@:chained_hash_keyed_set[`Key,`Value],
                             c:&(Value):void):void;
  method includes(t@:chained_hash_keyed_set[`Key,`Value <= comparable[Value]],
                  x:Value):bool;
  method fetch(t@:chained_hash_keyed_set[`Key,`Value], key:Key,
               if_absent:&():Value):Value;
  method match(t@:chained_hash_keyed_set[`Key,`Value], v:Value,
               if_absent:&():Value):Value;
  method add(t@:chained_hash_keyed_set[`Key,`Value], x:Value):void;
  method add_nonmember(t@:chained_hash_keyed_set[`Key,`Value], x:Value):void;
  method remove_key(t@:chained_hash_keyed_set[`Key,`Value], key:Key,
                    if_absent:&():`Else):Value|Else;
  method remove_match(t@:chained_hash_keyed_set[`Key,`Value], x:Value,
                      if_absent:&():`Else):Value|Else;
  method remove(t@:chained_hash_keyed_set[`Key,`Value <= comparable[Value]],
                x:Value, if_absent:&():void):void;
  method remove_any(t@:chained_hash_keyed_set[`Key,`Value],
                    if_empty:&():`Else):Value|Else;
  method remove_all(t@:chained_hash_keyed_set[`Key,`Value]):void;
  method remove_if(t@:chained_hash_keyed_set[`Key,`Value],
                   pred:&(Value):bool):int;
  method remove_keys_if(t@:chained_hash_keyed_set[`Key,`Value],
                         pred:&(Key):bool):int;
  method new_chained_hash_keyed_set[Key <= hashable[Key],
                                    Value <= keyed_hashable[Key]](
        ):chained_hash_keyed_set[Key,Value];
  method new_chained_hash_keyed_set[Key <= hashable[Key],
                                    Value <= keyed_hashable[Key]](
        size:int):chained_hash_keyed_set[Key,Value];
  method copy_empty(c@:chained_hash_keyed_set[`Key,`Value])
```

```
    :chained_hash_keyed_set[Key,Value];
```

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**. **chained_hash_keyed_set** is an implementation of mutable keyed sets using closed hashing; **hash_keyed_set** is an implementation of mutable keyed sets based on open 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.

In `hash-keyed-set.cecil`:

```
template object hash_keyed_set[Key <= hashable[Key],
                               Value <= keyed_hashable[Key]]
                    isa m_keyed_set[Key,Value], open_table[Value];
method collection_name(@:hash_keyed_set[`Key,`Value]):string;
method do(t@:hash_keyed_set[`Key,`Value], c:&(Value):void):void;
method do_allowing_updates(t@:hash_keyed_set[`Key,`Value],
                           c:&(Value):void):void;
method includes(t@:hash_keyed_set[`Key,`Value <= comparable[Value]],
                val:Value):bool;
method fetch(t@:hash_keyed_set[`Key,`Value], key:Key,
             if_absent:&():Value):Value;
method match(t@:hash_keyed_set[`Key,`Value], val:Value,
             if_absent:&():Value):Value;
method add(t@:hash_keyed_set[`Key,`Value], val:Value):void;
method remove_key(t@:hash_keyed_set[`Key,`Value], key:Key,
                  if_absent:&():`Else):Value|Else;
method remove_match(t@:hash_keyed_set[`Key,`Value], val:Value,
                       if_absent:&():`Else):Value|Else;
method remove(t@:hash_keyed_set[`Key,`Value <= comparable[Value]], val:Value,
              if_absent:&():void):void;
method remove_any(t@:hash_keyed_set[`Key,`Value],
                  if_empty:&():`Else):Value|Else;
method remove_all(t@:hash_keyed_set[`Key,`Value]):void;
method remove_if(t@:hash_keyed_set[`Key,`Value], pred:&(Value):bool):int;
method remove_keys_if(t@:hash_keyed_set[`Key,`Value],
                       pred:&(Key):bool):int;
let var default_hash_keyed_set_size:int;
method new_hash_keyed_set[Key <= hashable[Key], Value <= keyed_hashable[Key]](
      ):hash_keyed_set[Key,Value];
method new_hash_keyed_set[Key <= hashable[Key], Value <= keyed_hashable[Key]](
      size:int):hash_keyed_set[Key,Value];
method copy_empty(c@:hash_keyed_set[`Key,`Value]):hash_keyed_set[Key,Value];
method copy(t@:hash_keyed_set[`Key,`Value]):hash_keyed_set[Key,Value];
```

4.11 Collectors

In `collector.cecil`:

```
extend collector[`T] isa ordered_collection[T];
      extend type collector[T] subtypes collector[T1];
method flat_vector(c@:collector[`T <= sequence[`S]]):vector[S];
```

```
method add_first(c@:collector[`T], x:`T):void;
method add_last(c@:collector[`T], x:T):void;
method new_collector[T <= sequence[`S]]():collector[T];
method new_collector[T <= sequence[`S]](size:int):collector[T];
method copy(c@:collector[`T]):collector[T];
method collection_name(@:collector[`T]):string;
signature flat_string(string|collector[string]):string;
method flat_string(c@:collector[string]):string;
method flat_string(c@:string):string;
signature write(unix_file, indexed[char] | collector[indexed[char]],
                if_error:&(int):void):void;
method write(f@:unix_file, c@:collector[indexed[char]],
             if_error:&(int):void):void;
signature write(unix_file, indexed[char] | collector[indexed[char]]):void;
method write(f@:unix_file, c@:collector[indexed[char]]):void;
signature &&((`T <= sequence[`S]) | collector[`T],
             (`T <= sequence[`S]) | collector[`T]):collector[T];
method &&(s1@sequence[`S]:`T <= sequence[`S],
          s2@sequence[`S]:`T <= sequence[`S]):collector[T];
method &&(s1@sequence[`S]:`T <= sequence[`S], c@:collector[`T]):collector[T];
method &&(c@:collector[`T], s2@sequence[`S]:`T <= sequence[`S]):collector[T];
method &&(c1@:collector[`T], c2@:collector[`T]):collector[T];
method ||(c1@:collector[`T], c2@:collector[`T]):collector[T];
```

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"
```

4.12 Histograms

In `histogram.cecil`:

```
let var max_histogram_values_to_keep:int;
template object histogram[T <= ordered_hashable[T]]
                                            isa hash_CR_table[T,integer];
  field title(@:histogram[`T]):string;
  field individual_values(@:histogram[`T]):hash_table[T,m_bag[any]];
  method new_histogram[T <= ordered_hashable[T]]():histogram[T];
  method new_histogram[T <= ordered_hashable[T]](t:string):histogram[T];
  method increment(t@:histogram[`T], x:T):void;
  method increment_by_count(t@:histogram[`T], x:T, cnt:integer):void;
  method add_value(t@:histogram[`T], k:T, elem:any):void;
  method increment(t@:histogram[`T], x:T, elem:any):void;
  method print_statistics(t@:histogram[`T]):string;
  method print_statistics(t@:histogram[`T], some_key:T|string):string;
  method print_statistics(t@:histogram[`T <= num], some_key@num:T):string;
```

```
  method sort_it(t@:histogram[`T]):m_sorted_collection[T];
  method print_string(t@:histogram[`T]):string;
  method print(t@:histogram[`T]):void;
  method frequency_sorted_print_string(t@:histogram[`T]):string;
  method percent_print_string(t@:histogram[`T]):string;
  method truncated_percent_print_string(t@:histogram[`T], over:T):string;
  method distribution[`T <= ordered_hashable[T]](
        nm:string, cl:&(increment:&(value:T):void,
                        add_value:&(value:T, elem:any):void):void
        ):histogram[T];
  method distribution(nm:string,
                      cl:&(increment:&(value:int):void,
                           add_value:&(value:int, elem:any):void):void
                      ):histogram[int];
```

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 and its sorted print_string output, the values being counted by the `histogram` must be both hashable and ordered.

4.13 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.cecil`:

```
template representation filtered_table[Key,Value,Table <= table[Key,Value]]
                        isa table[Key,Value];
method length(t@:filtered_table[`Key,`Value,`Table]):int;
method do_associations(t@:filtered_table[`Key,`Value,`Table],
                       c:&(Key,Value):void):void;
method fetch(t@:filtered_table[`Key,`Value,`Table], key:Key,
             if_absent:&():Value):Value;
method collection_name(@:filtered_table[`Key,`Value,`Table]):string;
template representation m_filtered_table[Key,Value,Table <= m_table[Key,Value]]
                        isa filtered_table[Key,Value,Table], m_table[Key,Value];
method store(t@:m_filtered_table[`Key,`Value,`Table], key:Key, value:Value,
             if_absent:&():void):void;
method view_filtered(t@:table[`Key,`Value], filter:&(Key):bool)
                    :table[Key,Value];
method view_filtered(t@:m_table[`Key,`Value], filter:&(Key):bool)
                    :m_table[Key,Value];
method view_subset(t@:table[`Key <= comparable[Key],`Value], keys@:set[Key])
                  :table[Key,Value];
method view_subset(t@:m_table[`Key <= comparable[Key],`Value], keys@:set[Key])
                  :m_table[Key,Value];
```

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.

In `mapped.cecil`:

```
template representation mapped_table[Key1, Key2, Value]
                        isa table[Key1, Value];
method length(v@:mapped_table[`Key1, `Key2, `Value]):int;
method is_empty(v@:mapped_table[`Key1, `Key2, `Value]):bool;
method fetch(v@:mapped_table[`Key1, `Key2, `Value], key:Key1,
             if_absent:&():Value):Value;
method collection_name(@:mapped_table[`Key1, `Key2, `Value]):string;
template representation m_mapped_table[Key1, Key2, Value]
                isa mapped_table[Key1,Key2,Value], m_table[Key1, Value];
method store(v@:m_mapped_table[`Key1, `Key2, `Value], key:Key1, value:Value,
             if_absent:&():void):void;
method view_mapped(t@:table[`Key2, `Value], map@:table[`Key1, Key2])
                  :table[Key1, Value];
method view_mapped(t@:m_table[`Key2, `Value], map@:table[`Key1, Key2])
                  :m_table[Key1, Value];
method view_mapped(t@:mapped_table[`Key2, `Key3, `Value],
                   map@:table[`Key1, Key2]):table[Key1, Value];
signature view_mapped(m_table[`Key2, `Value],
                      map:table[`Key1, Key2]):m_table[Key1, Value];
signature view_mapped(table[`Key2, `Value],
                      map:indexed[Key2]):indexed[Value];
signature view_mapped(m_table[`Key2, `Value],
                      map:indexed[Key2]):m_indexed[Value];
template representation indexed_table[Key, Value]
    isa mapped_table[int,Key,Value], indexed[Value];
template representation m_indexed_table[Key, Value]
    isa indexed_table[Key,Value],
        m_mapped_table[int,Key,Value],
        m_indexed[Value];
method view_index_mapped(t@:table[`Key, `Value], map@:indexed[Key])
                        :indexed[Value];
method view_index_mapped(t@:m_table[`Key, `Value], map@:indexed[Key])
                        :m_indexed[Value];
method view_index_mapped(t@:indexed_table[`Key, `Value], map@:indexed[int])
                        :indexed[Value];
method view_index_mapped(t@:m_indexed_table[`Key, `Value], map@:indexed[int])
                        :m_indexed[Value];
method view_subrange(t@:indexed[`T], from:int):indexed[T];
method view_subrange(t@:m_indexed[`T], from:int):m_indexed[T];
method view_subrange(t@:indexed[`T], from:int, to:int):indexed[T];
method view_subrange(t@:m_indexed[`T], from:int, to:int):m_indexed[T];
method view_subrange(t@:indexed[`T], from:int, to:int, step:int):indexed[T];
method view_subrange(t@:m_indexed[`T], from:int, to:int, step:int)
                    :m_indexed[T];
```

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);
```

```
template representation string_view isa indexed_table[int,char], string;
template representation m_string_view
                        isa string_view, m_string, m_indexed_table[int,char];
method view_string_index_mapped(s@:string, map@:indexed[int]):string;
method view_string_index_mapped(s@:m_string, map@:indexed[int]):m_string;
method view_string_index_mapped(s@:string_view, map@:indexed[int]):string;
method view_string_index_mapped(s@:m_string_view, map@:indexed[int]):m_string;
method view_subrange(t@:string, from:int):string;
method view_subrange(t@:m_string, from:int):m_string;
method view_subrange(s@:string, from:int, to:int):string;
method view_subrange(s@:m_string, from:int, to:int):m_string;
method view_subrange(s@:string, from:int, to:int, step:int):string;
method view_subrange(s@:m_string, from:int, to:int, step:int):m_string;
method compose(t1@:table[`Key2,`Value], t2@:table[`Key1,Key2])
             :table[Key1,Value];
method compose(t1@:table[`Key2,`Value], t2@:indexed[Key2])
             :indexed[Value];
method compose(t1@:interval, t2@:interval):interval;
```

Views of strings preserve "stringness."

5 Input/output

5.1 Streams

In stream.cecil:

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.

```
abstract object stream[T];
signature is_at_end(stream[`T]):bool;
method before_end(s@:stream[`T]):bool;
method forward(s@:stream[`T]):void;
signature forward(stream[`T], at_end:&():void):void;
implementation forward(s@:stream_before_end[`T], at_end:&():void):void;
implementation forward(s@:stream_at_end[`T], at_end:&():void):void;
signature next(stream[`T], at_end:&():T):T;
implementation next(s@:stream_before_end[`T], at_end:&():T):T;
implementation next(s@:stream_at_end[`T], at_end:&():void):void;
method next(s@:stream[`T]):T;
method next_N(s@:stream[`T], n:int):sequence[T];
```

```
signature peek_next(stream[`T], at_end:&():T):T;
implementation peek_next(s@:stream_at_end[`T], at_end:&():T):T;
method peek_next(s@:stream[`T]):T;
method as_collection(s@:stream[`T]):sequence[T];
```

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 object m_stream[T] isa stream[T];
signature set_peek_next(m_stream[`T], :T):void;
implementation set_peek_next(@:m_stream_at_end[`T], :T):void;
method set_next(s@:m_stream[`T], x:T):void;
method flush(@:m_stream[`T]):void;
```

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 object removable_stream[T] isa stream[T];
signature remove_next(removable_stream[`T]):void;
implementation remove_next(@:removable_stream_at_end[`T]):void;
```

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

```
abstract object insertable_stream[T] isa stream[T];
signature insert(insertable_stream[`T], :T):void;
method insert_all(s@:insertable_stream[`T], c@:ordered_collection[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 object extensible_stream[T] isa stream[T];
signature add_last(extensible_stream[`T], x:T):void;
method add_all_last(s@:extensible_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 object reversible_stream[T] isa stream[T];
signature is_at_start(reversible_stream[`T]):bool;
method after_start(s@:reversible_stream[`T]):bool;
signature backward(reversible_stream[`T], at_start:&():void):void;
implementation backward(s@:reversible_stream_after_start[`T],
                        at_start:&():void):void;
```

```
implementation backward(s@:reversible_stream_at_start[`T],
                        at_start:&():void):void;
method backward(s@:reversible_stream[`T]):void;
signature prev(reversible_stream[`T], at_start:&():T):T;
implementation prev(s@:reversible_stream_after_start[`T], at_start:&():T):T;
implementation prev(s@:reversible_stream_at_start[`T], at_start:&():T):T;
method prev(s@:reversible_stream[`T]):T;
signature peek_prev(reversible_stream[`T], at_start:&():T):T;
implementation peek_prev(@:reversible_stream_at_start[`T], at_start:&():T):T;
method peek_prev(s@:reversible_stream[`T]):T;
abstract object m_reversible_stream[T] isa reversible_stream[T], m_stream[T];
signature set_peek_prev(m_reversible_stream[`T], :T):void;
implementation set_peek_prev(@:m_reversible_stream_at_start[`T], :T):void;
method set_prev(s@:m_reversible_stream[`T], x:T):void;
```

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 object positionable_stream[T] isa reversible_stream[T];
signature position(positionable_stream[`T]):int;
signature set_position(positionable_stream[`T], :int, off_end:&():void):void;
method set_position(s@:positionable_stream[`T], p:int):void;
method forward(s@:positionable_stream[`T], at_end:&():void):void;
method backward(s@:positionable_stream[`T], at_start:&():void):void;
method to_start(s@:positionable_stream[`T]):void;
method to_end(s@:positionable_stream[`T]):void;
method is_at_start(s@:positionable_stream[`T]):bool;
method is_at_end(s@:positionable_stream[`T]):bool;
signature length(positionable_stream[`T]):int;
method is_empty(s@:positionable_stream[`T]):bool;
method non_empty(s@:positionable_stream[`T]):bool;
method view_indexed(@:positionable_stream[`T]):indexed[T];
abstract object m_positionable_stream[T] isa positionable_stream[T],
                                             m_reversible_stream[T];
```

A **positionable_stream** supports querying and setting the position in the stream explicitly. The length of the stream can be determined also. Finally, a positionable stream can be viewed as an indexed collection using the view_indexed operation. (view_indexed may not be implemented yet.)

5.2 stream_views ; view_stream

In `indexed-stream.cecil`:

```
template representation indexed_stream_view[T] isa positionable_stream[T];
  method position(v@:indexed_stream_view[`T]):int;
  method set_position(v@:indexed_stream_view[`T], p:int,
                      off_end:&():void):void;
  method peek_next(v@:indexed_stream_view[`T], at_end:&():T):T;
  method peek_prev(v@:indexed_stream_view[`T], at_start:&():T):T;
  method length(v@:indexed_stream_view[`T]):int;
```

```
  method view_stream(c@:indexed[`T]):positionable_stream[T];
template representation m_indexed_stream_view[T] isa m_positionable_stream[T],
                                               indexed_stream_view[T];
  method set_peek_next(v@:m_indexed_stream_view[`T], x:T):void;
  method set_peek_prev(v@:m_indexed_stream_view[`T], x:T):void;
  method view_stream(c@:m_indexed[`T]):m_positionable_stream[T];
```

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

In `list-stream.cecil`:

```
template object list_stream_view[T] isa m_stream[T],
                              removable_stream[T], insertable_stream[T];
method view_stream(l@:simple_list[`T]):list_stream_view[T];
method is_at_end(s@:list_stream_view[`T]):bool;
method is_at_start(s@:list_stream_view[`T]):bool;
method forward(s@:list_stream_view_before_end[`T], at_end:&():void):void;
method peek_next(s@:list_stream_view_before_end[`T], at_end:&():T):T;
method set_peek_next(s@:list_stream_view_before_end[`T], x:T):void;
method remove_next(s@:list_stream_view_before_end[`T]):void;
method insert(s@:list_stream_view[`T], x:T):void;
template representation m_list_stream_view[T] isa list_stream_view[T];
method view_stream(l@:m_list[`T]):m_list_stream_view[T];
method insert(s@:m_list_stream_view_at_start[`T], x:T):void;
```

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

5.3 Random numbers

Random numbers are supported through the **random_stream** interface.

In `randstream.cecil`:

```
template object random_stream isa stream[int];
method peek_next(t@:random_stream, at_end:&():int):int;
method next(t@:random_stream, at_end:&():int):int;
method is_at_end(@:random_stream):bool;
method new_rand_stream(i:int):random_stream;
method new_rand_stream(i:int, j:int):random_stream;
method random_vector(len:int, range:int):vector[int];
```

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.

5.4 Unix files

In `file.cecil`:

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

```
template object unix_file
          isa m_positionable_stream[char], extensible_stream[char];
```

```
method stdin():unix_file;
method stdout():unix_file;
method stderr():unix_file;
abstract object open_mode;
concrete representation open_for_reading isa open_mode;
concrete representation create_for_writing isa open_mode;
concrete representation open_for_append isa open_mode;
concrete representation open_for_update isa open_mode;
concrete representation create_for_update isa open_mode;
concrete representation open_for_appending_update isa open_mode;
concrete representation open_binary_for_reading isa open_mode;
concrete representation create_binary_for_writing isa open_mode;
concrete representation open_binary_for_append isa open_mode;
concrete representation open_binary_for_update isa open_mode;
concrete representation create_binary_for_update isa open_mode;
concrete representation open_binary_for_appending_update isa open_mode;
method open_file(n@:string, m@:open_mode):unix_file;
method open_file(n@:string, m@:open_mode, if_error:&(int):unix_file):unix_file;
method name(f@:unix_file):string;
method is_readable(f@:unix_file):bool;
method is_unreadable(f@:unix_file):bool;
method is_writable(f@:unix_file):bool;
method is_unwritable(f@:unix_file):bool;
method is_read_write(f@:unix_file):bool;
method is_append(f@:unix_file):bool;
method read(f@:unix_file, buffer@:m_indexed[char], size:int):int;
method read(f@:unix_file, buffer@:m_indexed[char], size:int,
            if_error:&(int):int):int;
method read(f@:unix_file, buffer@:m_vstring, size@:int,
            if_error:&(int):int):int;
method read_line(f@:unix_file, if_eof@closure:&():string):string;  -- returns the line's con-
tents with no trailing
n; ‘‘’’ if the line is empty
method read_line(f@:unix_file, if_eof@closure:&():string,
                               if_error@closure:&(i:int):string):string;
method read_line(f@:unix_file, buffer@:m_indexed[char], size:int):int;
method read_line(f@:unix_file, buffer@:m_indexed[char], size:int,
                 if_error:&(int):int):int;
method read_line(f@:unix_file, buffer@:m_vstring, size:int,
                 if_error:&(int):int):int;
method write_char(f@:unix_file, x:char):void;
method write_line(f@:unix_file):void;
method write_line(f@:unix_file, buffer@:indexed[char]):void;
method write(f@:unix_file, buffer@:indexed[char]):void;
method write(f@:unix_file, buffer@:indexed[char], size:int):void;
method write(f@:unix_file, buffer@:indexed[char], size:int,
                  if_error:&(int):void):void;
method write(f@:unix_file, buffer@:vstring, size@:int,
                  if_error:&(int):void):void;
method print(s@:indexed[char], f:unix_file):void;
method write_to_file(o:any, fname:string):void;
method write_to_file(o:any, fname:string, if_error:&(int):none):void;
```

```
method position(f@:unix_file):int;
method position(f@:unix_file, if_error:&(int):int):int;
abstract object position_mode;
concrete representation from_start isa position_mode;
concrete representation from_current_position isa position_mode;
concrete representation from_end isa position_mode;
method set_position(f@:unix_file, offset:int, if_error:&(int):void):void;
method set_position_relative(f@:unix_file, offset:int,
                             from@:position_mode):void;
method set_position_relative(f@:unix_file, offset:int, from@:position_mode,
                             if_error:&(int):void):void;
method detected_eof(f@:unix_file):bool;
method length(f@:unix_file):int;
method flush(f@:unix_file):void;
method flush(f@:unix_file, if_error:&(int):void):void;
method close(f@:unix_file):void;
method close(f@:unix_file, if_error:&(int):void):void;
method get_mod_time(f_name@:string, if_error:&(int):int):int;
method get_mod_time_internal(f_name@:string, if_error:&(int):int):int;
method mod_time(f@:unix_file):int;
method mod_time(f@:unix_file, if_error:&(int):int):int;
method error_string(i:int):string;
method unix_error(i:int, s:string):none;
method nonfatal_unix_error(i:int, s:string):void;
extend unix_file isa m_positionable_stream[char], extensible_stream[char];
method next(f@:unreadable_unix_file, at_end:&():char):char;
method next(f@:readable_unix_file, at_end:&():char):char;
method peek_next(f@:unix_file, at_end:&():char):char;
method peek_prev(f@:unix_file, at_start:&():char):char;
method prev(f@:unix_file, at_start:&():char):char;
method set_next(f@:unwritable_unix_file, x:char):void;
method set_next(f@:writable_unix_file, x:char):void;
method set_peek_next(f@:unix_file, x:char):void;
method set_peek_prev(f@:unix_file, x:char):void;
method set_prev(f@:unix_file, x:char):void;
method add_last(f@:unwritable_unix_file, x:char):void;
method add_last(f@:writable_unix_file, x:char):void;
```

Unix 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 Unix error during the operation, passing the Unix `errno` value as the integer argument to the closure. The `error_string` function converts `errno` to an error message, using the Unix `sys_errlist`. The `unix_error` function invokes error with an appropriate error message derived from a user-supplied message and the `errno` code. The `nonfatal_unix_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` functions read up to `size` characters into a buffer; they return how many characters actually were 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 `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, `unix_file`s 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.

```
method file_exists(s:string):bool;
method parse_path(path_string@:string, path_separator:char):extensible_sequence[string];
method parse_path(path_string@:string):extensible_sequence[string];
method path_name(t@:string):string;
method strip_leading_path(t@:string):string;
method dirname(t@:string):string;
method basename(t@:string):string;
method expand_filename(s@:string):string;
method expand_filename(s@:string, if_error:&(int):string):string;
method shrink_filename(s@:string):string;
method is_abs_filename(s@:string):bool;
method find_file(base_name@:string, dirs@:ordered_collection[string],
                 if_fail:&(string):string):string;
method find_file(base_name@:string, dirs@:sequence[string]):string;
method write_object_to_file_name(obj:any, f_name:string):void;
method write_object_to_file_name(obj:any, f_name:string,
                                 if_error:&(int):void):void;
method write_object_to_file(x:any, f@:unix_file, use_bs@:bool):void;
method write_object_to_file(x:any, f@:unix_file,
                            if_error@closure:&(int):void):void;
method write_object_to_file(x:any, f@:unix_file, use_bs@:bool,
                            if_error:&(int):void):void;
method read_object_from_file_name(f_name@:string):dynamic;
method read_object_from_file_name(f_name@:string, use_bs:bool):dynamic;
method read_object_from_file_name(f_name@:string, use_bs:bool,
                                  if_error:&(int):dynamic):dynamic;
method read_object_from_file(f@:unix_file, use_bs@:bool):dynamic;
method read_object_from_file(f@:unix_file,
                             if_error@closure:&(int):dynamic):dynamic;
method read_object_from_file(f@:unix_file, use_bs:bool,
                             if_error:&(int):dynamic):dynamic;
```

A number of file-related operations have been defined. The `find_file` operation takes a file name and a directory search path and returns an absolute path name for the first file that matches the name in the search path. To do this work `find_file` invokes `file_exists` to test whether a given file name is defined and expand to expand away ũser file-name prefixes. The `parse_path` helper function converts a Unix search path string (directory names separated by colons) into a sequence of directory names.

6 Miscellaneous

6.1 ask

In ask.cecil:

```
method ask(prompt@:string):string;
method ask_yes_no(prompt@:string):bool;
```

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.

6.2 Time and date

In time.cecil:

```
method current_time():int;
method real_time(cl:&():void):int;
template object date_info isa comparable[date_info];
  method new_date_info(time:int):date_info;
  method date():date_info;
  method =(d1@:date_info, d2@:date_info):bool;
  method seconds(d@:date_info):int;
  method minutes(d@:date_info):int;
  method hours(d@:date_info):int;
  method day_of_month(d@:date_info):int;
  method month_of_year(d@:date_info):int;
  method month_of_year_name(d@:date_info):string;
  method month_of_year_shortname(d@:date_info):string;
  method year(d@:date_info):int;
  method day_of_week(d@:date_info):int;
  method day_of_week_name(d@:date_info):string;
  method day_of_week_shortname(d@:date_info):string;
  method day_of_year(d@:date_info):int;
  method is_daylight_savings_time(d@:date_info):bool;
  method time_zone_name(d@:date_info):string;
  method print_string(d@:date_info):string;
```

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.

6.3 text_lines

In text-lines.cecil:

```
template object text_lines;
  field lines(@:text_lines):array[string];
```

```
  method new_text_lines(s:string):text_lines;
  method indent(tl@:text_lines, i:int):void;
  method as_collector(tl@:text_lines):collector[string];
  method as_string(tl@:text_lines):string;
```

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.

6.4 matrix

A matrix is a two-dimensional indexable collection.

In `matrix.cecil`:

```
abstract object matrix[T];
  extend type matrix[`T] subtypes matrix[`S >= T];
  signature num_rows(matrix[`T]):int;
  signature num_cols(matrix[`T]):int;
  signature fetch(matrix[`T], row:int, col:int):T;
  method row(m@:matrix[`T], row:int):indexed[T];
  method col(m@:matrix[`T], col:int):indexed[T];
  method rows(m@:matrix[`T]):indexed[indexed[T]];
  method indices_do(m@:matrix[`T], c:&(int,int):void):void;
  method do(m@:matrix[`T], c:&(int,int,T):void):void;
  method +(m1@:matrix[`T <= num], m2@:matrix[T]):matrix[T];
  method *(m1@:matrix[`T <= num], m2@:matrix[T]):matrix[T|int];
  method *_ugly(m1@:matrix[`T <= num], m2@:matrix[T]):matrix[T|int];
  extend matrix[`T <= comparable[T]] isa comparable[matrix[T]];
  method =(m1@:matrix[`T <= comparable[T]], m2@:matrix[`T]):bool;
  method print_string(m@:matrix[`T]):string;
  method copy_init(m@:matrix[`T], num_rows:int, num_cols:int,
                   init:&(int,int):T):matrix[T];
  signature copy_init[T](matrix[T], num_rows:int, num_cols:int,
                         init:&(int,int):T):matrix[T];
  method copy_mutable_init(m@:matrix[`T], num_rows:int, num_cols:int,
                           init:&(int,int):T):m_matrix[T];
  signature copy_mutable_init[T](matrix[T], num_rows:int, num_cols:int,
                                 init:&(int,int):T):m_matrix[T];
abstract object m_matrix[T] isa matrix[T];
  signature store(matrix[`T], row:int, col:int, value:T):void;
template representation vector_matrix[T] isa m_matrix[T];
  field rows(@:vector_matrix[`T]):m_vector[m_vector[T]];
  method num_rows(m@:vector_matrix[`T]):int;
  method num_cols(m@:vector_matrix[`T]):int;
  method fetch(m@:vector_matrix[`T], row:int, col:int):T;
  method store(m@:vector_matrix[`T], row:int, col:int, value:T):void;
  method row(m@:vector_matrix[`T], row:int):indexed[T];
  method new_vector_matrix_init[T](num_rows:int, num_cols:int,
                                   init:&(int,int):T):m_matrix[T];
  method copy_init[T](m@:vector_matrix[T], num_rows:int, num_cols:int,
```

```
                    init:&(int,int):T):matrix[T];
  method copy_mutable_init[T](m@:vector_matrix[T], num_rows:int, num_cols:int,
                            init:&(int,int):T):m_matrix[T];
```

Matrices support querying the number of rows and columns of a matrix, extracting an element, a row, or a column of the matrix, and converting the matrix into a vector of row vectors. 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.

6.5 Graphs and partial orders

In `graph.cecil`:

```
abstract object graph_node[Node <= graph_node[Node,Edge],
                           Edge <= graph_edge[Node,Edge]]
                isa comparable[graph_node[Node,Edge]];
  field in_edges (@:graph_node[`Node,`Edge]):m_set[Edge];
  field out_edges(@:graph_node[`Node,`Edge]):m_set[Edge];
  signature table_key(graph_node[`Node,`Edge]):string;
abstract object 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;
  method =(e1@:graph_edge[`Node,`Edge], e2@:graph_edge[Node,Edge]):bool;
  method add_edge(e@graph_edge[`Node,`Edge]:`Edge <= graph_edge[`Node,`Edge]
                 ):void;
  method remove_edge(e@graph_edge[`Node,`Edge]:`Edge <= graph_edge[`Node,`Edge]
                    ):void;
abstract object graph[Node <= graph_node[Node,Edge],
                      Edge <= graph_edge[Node,Edge]];
  var field nodes(@:graph[`Node,`Edge]):m_removable_table[string,Node];
  method add_node(g@:graph[`Node,`Edge], node:Node):void;
  method remove_node(g@:graph[`Node,`Edge], node:Node):void;
  method add_edge(g@:graph[`Node,`Edge], e:Edge):void;
  method remove_edge(g@:graph[`Node,`Edge], e:Edge):void;
  method print_header(g@:graph[`Node,`Edge]):string;
  method print_headers(g@:graph[`Node,`Edge]):string;
  method print_string(g@:graph[`Node,`Edge]):string;
  method print(g@:graph[`Node,`Edge]):void;
```

In `partial-order.cecil`:

```
abstract object partial_order_node[Node <= partial_order_node[Node]]
                isa graph_node[Node,partial_order_edge[Node]],
                    partially_ordered[partial_order_node[Node]],
                    hashable[partial_order_node[Node]];
  var field marked(@:partial_order_node[`Node]):bool;
```

```
  method is_top(t@:partial_order_node[`Node]):bool;
  method is_bottom(t@:partial_order_node[`Node]):bool;
  method up_nodes_do(t@:partial_order_node[`Node],
                    bl:&(Node):void):void;
  method down_nodes_do(t@:partial_order_node[`Node],
                      bl:&(Node):void):void;
  method traverse_up(t@partial_order_node[`Node]
                     :`Node <= partial_order_node[Node],
                    cl:&(Node):void):void;
  method traverse_down(t@partial_order_node[`Node]
                       :`Node <= partial_order_node[Node],
                      cl:&(Node):void):void;
  method order_print_string(t@:partial_order_node[`Node]):string;
template object partial_order_edge[Node <= partial_order_node[Node]]
                isa graph_edge[Node,partial_order_edge[Node]];
  method new_partial_order_edge(f@partial_order_node[`Node]
                                :`Node <= partial_order_node[Node],
                               t:Node):partial_order_edge[Node];
template object partial_order[Node <= partial_order_node[Node]];
  field nodes(@:partial_order[`Node]):m_set[Node];
  field tops(@:partial_order[`Node]):m_set[Node];
  field bottoms(@:partial_order[`Node]):m_set[Node];
  method add_node(t@:partial_order[`Node], node:Node):void;
  method add_partial_order_edges(t@:partial_order[`Node]):void;
  method add_edge(t@:partial_order[`Node], e:partial_order_edge[Node]):void;
  method remove_edge(t@:partial_order[`Node], e:partial_order_edge[Node]):void;
  method remove_node(t@:partial_order[`Node], node:Node):void;
  method top_down_do(t@:partial_order[`Node], cl:&(Node):void):void;
  method bottom_up_do(t@:partial_order[`Node], cl:&(Node):void):void;
  method print_header(g@:partial_order[`Node]):string;
  method print_headers(t@:partial_order[`Node]):string;
  method print_string(t@:partial_order[`Node]):string;
  method print(t@:partial_order[`Node]):void;
  method clear_marks(g@:partial_order[`Node]):void;
  method new_partial_order[Node <= partial_order_node[Node]]
                         ():partial_order[Node];
```

6.6 System operations

In system.cecil:

```
method exit(error_code@:int):none;
method object_size(obj:any):int;
method individual_object_size(obj:any):indexed[int];
method object_size_histogram(obj:any):int;
method PIC_statistics():void;
method detailed_PIC_statistics():void;
method cpu_time():int;  -- Current CPU time in milliseconds.
method time(closure:&():void):int;  -- CPU execution time of closure in milliseconds
method benchmark_closure(cls:&():void):void;

method system(s@:string):int;
method system(s@:vstring):int;
```

```
method system(s@:string, if_error:&(i:int):int):int;
method breakpoint():void;
method sys_breakpoint():void;
concrete representation argv isa i_indexed[string];  -- Unix command-line arguments
method length(t@:argv):int;
method fetch(t@:argv, i:int, if_absent:&():string):string;
```

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.

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

Unix environment variables can be read and modified.

6.7 Window system

File `motif.cecil` provides an interface to the Motif window system toolbox. Its functions provide a variety of window system functionality, including creating frames, dialogs, buttons, lists, and text displays. See `motif.cecil` for the specific functions and their interfaces.

6.8 Message unparsing (reflection)

In `msg.cecil`:

```
method send(msg_name@:string, num_params@:int,
            args@:ordered_collection[dynamic]):dynamic;
method send(msg_name@:string, num_params@:int,
            args@:ordered_collection[dynamic], if_error:&():dynamic):dynamic;
method send(msg_name@:vstring, num_params@:int,
            args@:vector[dynamic], if_error:&():dynamic):dynamic;
method field_init_send(msg_name@:string, num_params@:int,
                       args@:ordered_collection[dynamic],
                       if_error:&():dynamic):dynamic;
method field_init_send(msg_name@:vstring, num_params@:int,
                       args@:vector[dynamic], if_error:&():dynamic):dynamic;
method prim_resend(msg_name@:string, num_params@:int,
                   args@:ordered_collection[dynamic],
```

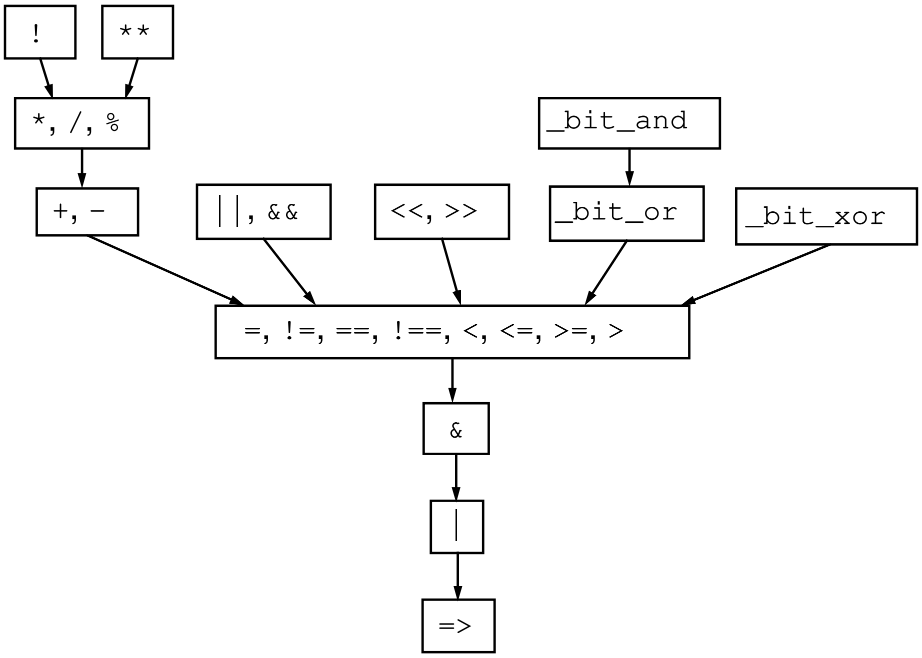

Figure 2: Precedence groups for binary operators; higher groups have higher precedence.

```
                 dirs@:ordered_collection[dynamic],
                 is_undirected:bool, if_error:&():dynamic):dynamic;
method prim_resend(msg_name@:vstring, num_params@:int,
                 args@:vector[dynamic], dirs@:vector[dynamic],
                 is_undirected:bool, if_error:&():dynamic):dynamic;
method directed_field_init_send(msg_name@:string, num_params@:int,
                                args@:ordered_collection[dynamic],
                                dirs@:ordered_collection[dynamic],
                                if_error:&():dynamic):dynamic;
method directed_field_init_send(msg_name@:vstring, num_params@:int,
                                args@:vector[dynamic], dirs@:vector[dynamic],
                                if_error:&():dynamic):dynamic;
method type_id(t:any):int;
method set_breakpoint(msg_name@:string):void;
method set_breakpoint(msg_name_oop@:vstring):void;
method show_breakpoints():void;
```

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

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.

Index