Vortex RTL Textual Description Grammar

The Cecil Group

Department of Computer Science and Engineering
University of Washington
Box 352350, Seattle, Washington 98195-2350
cecil@cs.washington.edu

This document describes the grammar for the textual form of the Vortex compiler's RTL intermediate language. This is the language used to communicate between the C++, Modula-3, Java, and Smalltalk front-ends and Vortex, and is also used to describe some Cecil primitives.

Warning: the RTL language was not designed from scratch, but rather evolved via accretion of features. It is slowly being reworked to make the operators more general and language-independent, but this process is not complete.

In our EBNF notation, vertical bars (`|`) are used to separate alternatives. Braces (`{...}`) surround strings that can be repeated zero or more times. Brackets (`[...]`) surround an optional string. Parentheses are used for grouping. Literal tokens are included in quotation marks (`"..."`).

1 Top-Level Entry Points

For Cecil RTL primitives, the syntax of the `code` string part of the `prim_body` is as follows:

```
rtl_prim        ::= rtl_stmts
```

For languages that have separate front-ends, such as Java, Smalltalk, C++, and Modula-3, source files are translated by the front-end into RTL files. An RTL file is described by the following grammar:

```
rtl_file        ::= { rtl_decl | pragma }
rtl_decl        ::= rtl_object_decl
                 |   rtl_extend_decl
                 |   rtl_method_decl
                 |   rtl_extern_method_decl
                 |   rtl_field_decl
                 |   rtl_var_decl
                 |   rtl_assoc_decl
                 |   rtl_include_decl
                 |   rtl_top_level_stmts
                 |   rtl_prim_stmts
```

The RTL file should not declare a method named `main` as this function is defined by the Vortex back-end. Instead, code to initialize RTL program data structures and invoke top-level procedures (like the equivalent of main) should be included as top-level statement blocks. The global variables `global_argc`, `global_argv`, and `global_envp` can be referenced to get at the usual `argc`, `argv`, and `envp` parameters of `main`, and the global variable `global_result` should be assigned to set the desired integer return code for `main`.

2 Global Declarations

Declarations at the top level can appear in any order, and can be mutually recursive. Evaluation of global variable initializers is done top-down in the order given.

An **object declaration** introduces a new class. Abstract classes cannot be instantiated directly. The `isa` list names its immediate superclasses. (There is a root class implicitly defined, and all classes inherit (directly or indirectly) from this root class implicitly.) Inheritance defines a partial order on objects; redundant or transitively-implied inheritance links are ignored. In the presence of diamond-shaped multiple inheritance, shared ancestor classes are including only once in the hierarchy, as if all classes were virtual base classes in C++ terminology.

```
rtl_object_decl ::= [rtl_obj_role] “object” name [“isa” names]
                        {pragma|object_id_pragma} “;”
names           ::= name {“,” name}
```

Objects can be `abstract` (and have no instances or direct references), `concrete` (allowing creation of instances as with the `new` operation but no direct references to the named object itself), or `unique` (allowing direct references by name to the object itself as its sole “instance”, and disallowing any `new` operations on the named object to create other instances). Subclassing from any of these kinds of objects is allowed.

```
rtl_obj_role    ::= “abstract” | “concrete” | “unique”
```

An `object_id_pragma` is attached to an `object_decl`, and specifies the names of the symbols to use as class identifiers for that class, when it is viewed from particular static types.

```
object_id_pragma ::= “(**” “ID” “(” “[” object_id_pair {“,” object_id_pair } “]”
                     “**)”
object_id_pair  ::= “[” static_type “,” name “]”
```

An **object extension declaration** adds one or more inheritance link to a previously declared or predefined object.

```
rtl_extend_decl ::= “extend” name “isa” names
```

A **method declaration** defines a procedure in the intermediate program.

```
rtl_method_decl ::= “method” [rtl_visibility] name
                      “(” {rtl_formals} “)” “:” representation {pragma}
                      “{” rtl_stmts “}” [“;”]
rtl_formals     ::= rtl_formal {“,” rtl_formal}
```

Each formal has a name (can be omitted if it’s not used within the body of the method), a specializer class (if omitted or if given as `@any`, then the argument is unspecialized), and a representation (if omitted, then it defaults to `OOP`). If `formals_are_immutable` is defined, then formal variables cannot be assigned, otherwise they can be. If `uses_PIC_based_runtime` is set, then only `OOP` formals can be specialized to something other than `any`. Formal parameters and results are passed by value (i.e. shallow-copied), including formals represented using `bytes[`*n*`]`, unlike C which passes arrays by reference.

```
rtl_formal      ::= [name] [“@” name] [“:” representation]
```

A method (or field or global variable declaration) can be marked `private` (meaning that it is scoped within the current file, as in C; `static` has the same meaning), or `extern` (meaning that it is visible to other files, as in C, and that it has been declared already in some other file).

```
rtl_visibility  ::= “private” | “static” | “extern”
```

An **extern method declaration** declares a method and says that its definition may be external to the part of the program that the compiler sees. It is allowable to provide an extern method declaration in one source file and the definition of the method in another source file. In this case, the actual definition of the method takes precedence over the extern declaration.

```
rtl_extern_method_decl::= “extern” “method” name “(” {rtl_formals} “)”
                          “:” representation {pragma} “;”
```

A **field declaration** declares an instance variable on a class. If var is present, the field is mutable, otherwise the field is considered immutable. The formal parameter specifies which class the instance variable is part of. If `front_end_does_field_layout` is set, then field declarations are unnecessary, and only low-level object allocation (with precomputed size) and pointer load & store operations are supported, otherwise field declarations should be provided and high-level object allocation operations and field load and store operations are provided. If `generate_field_accessors` is set, then a field declaration implicitly generates a get accessor method (and a set accessor method, if mutable).

```
rtl_field_decl  ::= [“var”] [array_field_info] “field” [rtl_visibility] name
                      “(” rtl_formal “)” “:” representation {pragma}
                      [“:=” literal] “;”
```

Some fields are special and are used for arrays. A field with the `array` annotation declares an *indexed* field holding zero or more copies of the field contents (the representation of an indexed field refers to the representation of each element of the array). A field with the `length` annotation holds the number of elements of the indexed field; this field is named by the indexed field. (Length fields currently should have representation `OOP`.) Optionally, the type of elements of the indexed field may be recorded at run-time, using a field annotated with `elem_type`. (Elem_type fields currently should have representation `CecilMap*`.) The array field should name the corresponding length field and the elem_type field, if present. At present, a class can have or inherit at most one indexed field.

```
array_field_info ::= “length”
                 |   “array” “of” name [“elem_type” name]
                 |   “elem_type”
```

A **global variable declaration** introduces a variable in the global (current) scope, with the given visibility and representation. Variables (global and local) are assumed to be assignable.

```
rtl_var_decl    ::= “decl” [rtl_visibility] representation name {pragma}
                      [“:=” (literal | static_init)] “;”
```

A static initializer provides the mechanism for specifying the static initial value of a global variable whose representation is an array of bytes. It provides a sequence of initializers, each of which specifies an offset, a representation, and an initial value of that representation placed at that offset. The different kinds of initial values are pad (uninitialized data), the address of some global variable plus the given offset, a procedure address (also specifying the return type of the procedure), an integer constant (possibly given in a string literal), a float constant (possibly given in a string literal), and a byte sequence.

```
static_init     ::= “static_init” “{” {static_init_entry} “}” “;”
static_init_entry::= “[” int “,” representation “,” static_init_value “]”
static_init_value::= “pad”
                 |   “&” name [“+” integer]
                 |   “&_proc” name representation
                 |   integer | “istr” “(” string “)”
                 |   float | double | “fstr” “(” string “)”
                 |   “str” “(” string “)”
```

An **associate declaration** treats the named method as a method in the given message set with the listed specializers.

```
rtl_assoc_decl  ::= “associate” name “with”
                      name “(” rtl_formals “)” “;”
```

For example:

```
   associate wrapper__3FooFi with wrapper__Fi(@Foo, @any);
```

This says consider the routine named `wrapper__3FooFi` to be an implementation of the message `wrapper__Fi` attached at class `Foo`.

An **include declaration** names another file of RTL code to include as part of the program.

```
rtl_include_decl  ::= “include” string “;”
```

A **top-level statement block** is a list of RTL statements that are to be executed upon program start-up. These expressions are evaluated in the order encountered during reading of the input files.

```
rtl_top_level_stmts ::= “rtl” “{” rtl_stmts “}” [“;”]
```

A **top-level primitive statements block** is an arbitrary block of code written in some external language (currently just C++) that is included in the compiled file at that point in the file.

```
rtl_prim_stmts    ::=  “prim” language “{” code “}” “;”
language          ::=  “c_++” | ...
code              ::=  arbitrary bracket-balanced code written in language
```

3 Data Representations

Representations are specified in declarations and in statements. They indicate the representation (format, bit encoding, size, etc.) of global variables, formals, method results, locals, and so on. A representation is an unstructured sequence of bytes (whose components are accessed through pointer loads & stores), optionally with some alignment, something that is roughly equivalent to one of C’s atomic data types (perhaps with a specified size, otherwise with a “natural” size), or something that is a special tagged-pointer-sized integer, float, tagged pointer, NLR return code, or return pair.

```
representation    ::=  “bytes” “[” integer “]” [ “align” integer ]
                  |    base_rep {“*”}
base_rep          ::=  “char” | “u_char” | “short” | “u_short”
                  |    “int”  | “u_int”  | “long”  | “u_long”
                  |    “float” | “double”
                  |    “int1” | “u_int1” | “int2” | “u_int2”
                  |    “int4”  | “u_int4”  | “int8”  | “u_int8”
                  |    “float4” | “float8”
                  |    “void”
                  |    “wordInt” | “unsignedWordInt” | “wordFloat” | “OOP”
                  |    “ReturnCode” | “PairType”
```

4 Statements

RTL statements are the following:

```
rtl_stmts         ::=  { rtl_stmt “;” }
rtl_stmt          ::=  alu
                  |    field_op
                  |    ptr_op
                  |    allocation
                  |    exception_scope
                  |    label
                  |    goto
                  |    fn_call
                  |    return
```

```
                |   fatal
                |   static_info
```

Many RTL statements may be prefaced with a declaration. These declarations indicate the representation of the variable. A given variable should be declared at the first textual place it is assigned or read, but the scope of the declaration is the entire method. A variable declared with the `source` keyword can be referenced from nested closures.

```
decl            ::=  “decl” [“source”] representation
```

Local variable declarations, assignments, unary and binary operations, and conversions between different numeric representations (which specify the target representation to convert to).

```
alu             ::=  local_decl | assign | unop | binop | conversion
local_decl      ::=  [decl] name
assign          ::=  [decl] name “:=” value
unop            ::=  [decl] name “:=” unop value
binop           ::=  [decl] name “:=” value binop value
conversion      ::=  [decl] name “:=” “convert” value representation
```

Arguments to RTL operators can name RTL variables, can name undeclared variables (in which case they are assumed to name global variables of type `void*` that are defined by the run-time system and the header files included with the generated code), can mention integer, long, float, double, character, or string literals, and can mention language-specific void, true, false, and null-pointer values, and can mention target-machine-specific constants for the word size and the number of tag bits.

```
value           ::=  name | literal | “void” | “true” | “false” | “null” |
                     “bytes_per_machine_word” |
                     “bits_per_machine_word” | “log_bits_per_machine_word” |
                     “num_tag_bits”
literal         ::=  integer | long | float | double | character | string
```

Field (instance variable) operations. Stores or fetches from a declared field of an object. Field contents are considered immutable when the corresponding field declaration specified an immutable field. If the field names an indexed field, then an index expression should be used to name the desired element of the indexed field (origin 0).

```
field_op        ::=  field_load | field_store
field_load      ::=  [decl] name “:=” field_location
field_store     ::=  [decl] field_location “:=” value
field_location  ::=  name “.” name “@” name [“[” index “]”]
index           ::=  name | integer
```

Simple pointer operations. Used to access language-level arrays, structs, and so on, if they’ve been translated down into this low level. The representation specifies the representation of the pointer target value being loaded or stored, and the index is a byte offset from the base pointer. Pointer target locations can be marked immutable, as with vector locations.

```
ptr_op          ::=  ptr_load | ptr_store | addr_op
ptr_load        ::=  [decl] name “:=” ptr_location
ptr_store       ::=  ptr_location “:=” value
ptr_location    ::=  “*” “(” name [“+” index] “)” representation
                        [“is_immutable”]
```

Pointers can be created by taking the address of a variable or of a procedure.

```
addr_op         ::=  [decl] name “:=” “&” name
```

```
                 |    [decl] name ":=" "&_proc" name
```

High-level allocation of new data structures. Object allocation can be used for classes with associated field declarations specifying the class's structure and `front_end_does_field_layout` turned off; various class-testing operations and field access operations can be applied to these objects. The object allocator allocates an instance of the given class; such classes should not be `unique`, `abstract`, `template`, or contain indexed fields. The array allocator is used to allocate objects with an indexed field, given the name of the class and the length of the array. The named class must satisfy the same restrictions as for an object allocation, but must contain an indexed field, and not contain an elem_type field. The typed array allocator is like an array allocator except that it instantiates classes that do contain an elem_type field. The vector allocator operations are array allocators for particular Cecil built-in classes. The closure allocator creates new lexically-nested anonymous closures, which are invoked using language-specific message names. `new_closure` is the normal closure allocation keyword; `new_LIFO_closure` can be used to allocate a closure that is known not to outlive its lexically-enclosing environment.

```
allocation       ::=  [decl] name ":=" allocator
allocator        ::=  obj_allocator
                 |    vec_allocator
                 |    array_allocator
                 |    typed_array_alloc
                 |    closure_allocator
obj_allocator    ::=  "new" name
array_allocator  ::=  "new_array" name value
typed_array_alloc ::="new_typed_array" name value static_type_spec
vec_allocator    ::=  vec_alloc_op value
vec_alloc_op     ::=  "new_m_vector" | "new_i_vector"
                 |    "new_m_string" | "new_i_string"
                 |    "new_m_float_vector" | "new_i_float_vector"
                 |    "new_m_word_vector" | "new_i_word_vector"
closure_allocator::=  closure_alloc_op "(" {rtl_formals} ")"
                        ":" representation "{" rtl_stmts "}"
closure_alloc_op ::=  "new_closure"
                 |    "new_LIFO_closure"
```

Exception scope control. Exceptions can be raised by calls, sends, and explicit raises. The scope of a handler for some exceptions is bracketed by `enter_` and `exit_exn_scope` RTL statements. These statements name the label where the innermost-enclosing exception handler for this scope begins. For languages where `exceptions_through_longjmp` is set, the runtime system is assumed to select the right handler if an exception is raised; the enter_exn_scope provides enough information to the runtime system to make this possible. Otherwise, exceptions propagate outwards through exception scopes incrementally, and each exception handling scope is responsible for forwarding control to the enclosing exception handling scope if it is not the right handler.

```
exception_scope  ::=  enter_exn_scope | exit_exn_scope
enter_exn_scope  ::=  "enter_exn_scope" label_name name ["(" name ")"]
                        ["[" value "," value "," value "]"]
exit_exn_scope   ::=  "exit_exn_scope" label_name
```

The `enter_exn_scope` operator implicitly declares the variable to which the "name" of the exception being raised will be bound, and optionally declares the name which will be bound to the argument of the raised exception; these names are expected to be used in the exception handling code. For Modula-3, three additional parameters are needed, giving the address of the exception record, the address of the module's exception handler list, and the class of the exception. (Other languages will presumably need extensions to support arguments appropriate for their runtime system implementation of exceptions.)

Labels. Labels that start exception handlers must be flagged as such.

```
label           ::= (“label” | “exn_handler_label”) label_name
label_name      ::= name
```

Various kinds of unconditional and conditional branches. `is_class` is true iff the value is an instance of the named class; `inherits_from` extends this test to include subclasses as well. `inherits_from` can take a variable name (a computed `CecilMap*`) as an argument instead of a class name constant. `check_typed_array(array,value,fld@obj)` implements the necessary checking to ensure that `value` is in the element type of typed array `array`, where the element type is stored in field `fld@obj`.

```
goto            ::= cond_goto | uncond_goto | case_goto | raise
cond_goto       ::= “if” test “goto” label_name
test            ::= unop value
                 |   value binop value
                 |   value “inherits_from” name
                 |   value “is_class” name
                 |   “check_typed_array”
                         “(” value “,” value “,” name “@” name “)”
uncond_goto     ::= “goto” label_name
```

`case_goto` implements an N-way branch. The argument value should be an integer in the range [0..N-1], and one of the N target labels is jumped to based on this value.

```
case_goto       ::= “case” value “goto” “[” label_names “]”
label_names     ::= label_name {“,” label_name}
```

Exceptions are raised using the `raise` command. The first operand evaluates to the exception “name” (or exception object, for e.g. C++ and Java), and the second optional value is the argument to the exception. Control transfers unconditionally to the appropriate enclosing exception handler when the raise is executed.

```
raise           ::= “raise_exn” value [“(” value “)”]
```

Various kinds of procedure calls. The result of a call can be omitted (e.g. if the callee returns no value). All kinds of calls allow a marker blocking inlining of the call as well as an indication of whether the call can raise exceptions (a.k.a. produce a non-local return). (If `sends_can_raise` is set, then sends, table sends, and calls to declared methods (as opposed to external procedures) default to raising exceptions.)

```
fn_call         ::= [[decl] name “:=”] (call | send | table_send)
                      [“no_inline”] [“can_raise_exns” | “no_exns”]
```

Calls are regular direct calls. The callee is either a regular named procedure (potentially one defined in RTL with a method decl, in which case it can be inlined; `call_external` blocks this check), or a value that evaluates to the address of the procedure to call.

```
call            ::= call_desc “(” [values] “)”
call_desc       ::= “call” name
                 |   “call_external” name
                 |   “call_indirect” value
values          ::= value {“,” value}
```

Sends are messages, and search (using Cecil's inheritance rules) the named method set for the most specific applicable method, based on the run-time classes of the send arguments. Send RTLs can only be used if `uses_PIC_based_runtime` is set.

```
send            ::=  "send" name "(" [values] ")"
```

Table sends are similar to sends, except that they get translated into the appropriate table lookup sequence (precomputed by the language front-ends) if they cannot be statically bound.

```
table_send      ::=  "table_send" name "(" [values] ")"
                       "[" table_index "]" [ "id_offset" "(" int_literal ")" ]
                       [static_type_info]
table_index     ::= int_literal
```

The address of a lookup table is fetched from the object specified as the first argument of the send, at the byte offset specified by the `id_offset` clause. If no `id_offset` is specified, the table address is assumed to be in the first word of the object. The `table_index` specifies the index into the table, giving a table entry to use for dispatching this send. For example:

```
   t1 := table_send my_message(t2, t3) [2] id_offset(4)
```

This means: fetch the address of the table from `table_base := *(t2+4)`, then use the table entry at `table_base+(2*sizeof(table_entry))` to dispatch this send. The precise way in which the table is used to dispatch the send is language dependent. Currently, Vortex supports two kinds of tables: a simple table with address-sized entries that contain the address of a routine to call, and a more complicated, C++-style table format that contains both the address of a routine and an adjustment value that should be added to the first argument before entering the callee.

The static type of the receiver argument of a table_send can be specified by naming the class that the receiver is known to inherit from.

```
static_type_info ::=  "static_type" "(" static_type ")"
static_type      ::=  name
```

Return statements. Non-local returns return from the lexically-enclosing non-closure method.

```
return          ::= ("return" | "non_local_return") [value]
```

Fatal errors. Abort execution with a string message.

```
fatal           ::=  "fatal" "(" string ")"
```

Assertions about the static properties of a value.

```
static_info     ::=  "assert_type" name static_type_info
```

Currently, static information specifies either an exact class for the variable, or specifies a cone of possible classes (and the value could be an instance of the class or any subclass of the class), or gives another variable whose type to copy. The optimizer uses this information to reason about the possible values that could be stored in the given variable.

```
static_type_info ::=  static_type_spec
                 |    "same_as" "(" name ")"
static_type_spec ::=  "cone" "(" name ")"
                 |    "map" "(" name ")"
                 |    "typed_array_map" "("
                         name "[" static_type_spec "]" ")"
```

5 Tokens

Bold-faced non-terminals in this grammar are the tokens in the grammar rules above. As usual, tokens are defined as the longest possible sequence of characters that are in the language defined by the grammar given below. The meta-notations "`one of "..."`", "`any but` *x*," and "*x*`..`*y*" are used to concisely list a range of alternative characters. `space`, `tab`, and `newline` stand for the corresponding characters.

Literals suffixed with a `p` or `P` character are boxed literals (for languages like Cecil and Smalltalk with boxed values), while unsuffixed literal tokens are unboxed literals.

A `long` is an 8-byte integer (e.g. a Java long value), while an `integer` is a word-sized integer. A `float` is an IEEE single-precision 4-byte float, while a `double` is an IEEE double-precision 8-byte float. An unboxed `character` literal is just an integer by another name. An unboxed `string` literal is a pointer to a statically-allocated array of characters. String literals are not null-terminated by default; an explicit `\0` should be added to the end of the string literal to get a terminating null character. Characters defined by `\` followed by 1-3 digits interpret the digits as an octal ascii code; decimal or hexadecimal interpretations can be selected by using the `d` or `x` prefix, respectively.

```
name            ::=  (letter | "_") {letter | digit | "_"}

integer         ::=  int_body ["p" | "P"]
int_body        ::=  ["-"] [radix] hex_digits
radix           ::=  digits "_"
hex_digits      ::=  hex_digit {hex_digit}
hex_digit       ::=  digit | one of "a..fA..F"

long            ::=  int_body ("l" | "L") ["p" | "P"]

float           ::=  float_body ["p" | "P"]
float_body      ::=  ["-"] integer "." hex_digits [exponent]
                |    ["-"] integer exponent
exponent        ::=  "^" ["+" | "-"] digits
digits          ::=  digit {digit}

double          ::=  float_body ("d" | "D") ["p" | "P"]

character       ::=  "'" char "'" ["p" | "P"]

string          ::=  """ { char | line_break } """ ["p" | "P"]
char            ::=  any | "\" escape_char
escape_char     ::=  one of "abfnrtv'"\?0"
                |    ["o"|"d"] digit [digit [digit]]
                |    "x" hex_digit [hex_digit]
line_break      ::=  "\" {whitespace} new_line {whitespace} "\"
```
characters between \'s are not part of the string

```
letter          ::=  one of "a..zA..Z"
digit           ::=  one of "0..9"
```

```
punct           ::=  one of “!#$%^&*-+=<>/?~\|”
```

6 Unary and Binary Operators

These are the unary and binary operators currently defined. For many operators, versions exist for different operand types, such as `+_int`, `+_unsigned_int`, `+_int8`, `+_unsigned_int8`. Such families of operators are notated in the table using brace-delimited lists over the possible types, such as `+_{int,unsigned_int,int8,unsigned_int8}`.

unop name	**description**
`unary_-_{int,int8, float,double}`	{integer, 8-byte integer, float, double} negation
`~_{int,int8}`	{integer, 8-byte integer} bitwise complement
`num_elems_int`	extract the number of elements of an array, vector, or string object.
`box_{int,char, float,double}`	convert an untagged machine {integer, integer, float, double} into a tagged Cecil {integer, character, float, double} OOP
`unbox_{int,char, float,double}`	inverse of `box_{int,char,float,double}`
`map_of`	returns the `CecilMap*` pointer representing the run-time class of the argument; may be used as the argument to `inherits_from`

binop name	**description**
`+_{int,unsigned_int, int8,unsigned_int8}`	Add two {signed integers, unsigned integers, signed 8-byte integers, unsigned 8-byte-integers}
`-_{int,unsigned_int, int8,unsigned_int8}`	Subtract ...
`*_{int,unsigned_int, int8,unsigned_int8}`	Multiply ...
`/_{int,unsigned_int, int8,unsigned_int8}`	Divide ...
`%_{int,unsigned_int, int8,unsigned_int8}`	Modulus ...
`&_{int,unsigned_int, int8,unsigned_int8}`	Bitwise and two {signed integers, unsigned integers, 8-byte-integers, unsigned 8-byte integers}
`|_{int,unsigned_int, int8,unsigned_int8}`	... or ...
`^_{int,unsigned_int, int8,unsigned_int8}`	... xor ...
`xnor_{int,unsigned_int, int8, unsigned_int8}`	... xnor ...
`<<_{int,unsigned_int, int8,unsigned_int8}`	bitwise left shift one {signed integer, unsigned integer, 8-byte-integer, unsigned 8-byte integer} by another

binop name	description
`>>_{int,unsigned_int, int8,unsigned_int8}`	... arithmetic right shift ...
`>>_logical_{int,int8}`	... logical right shift ...
`+_ptr`	Add two values, one of which is assumed to be a pointer and the other an integer byte count, to produce a new pointer
`-_ptr`	Subtract an integer byte count from a pointer to produce a new pointer
`-_ptrdiff`	Subtract two pointers to produce an integer difference, in bytes
`+_{float,double}`	Add two {floats, doubles}
`-_{float,double}`	Subtract ...
`*_{float,double}`	Multiply ...
`/_{float,double}`	Divide ...
`=_{int,unsigned,int8, unsigned_int8}_log`	Test whether two {signed integers, unsigned integers, 8-byte-integers, unsigned 8-byte integers} are equal
`!=_{int,unsigned,int8, unsigned_int8}_log`	... not equal
`>_{int,unsigned,int8, unsigned_int8}_log`	Test whether one {signed integer, unsigned integer, 8-byte-integer, unsigned 8-byte integer} is greater than another
`>=_{int,unsigned,int8, unsigned_int8}_log`	... greater than or equal to ...
`<_{int,unsigned,int8, unsigned_int8}_log`	... less than ...
`<=_{int,unsigned,int8, unsigned_int8}_log`	... less than or equal to ...
`=_ptr_log`	Test whether two pointers are equal
`!=_ptr_log`	... not equal
`>_ptr_log`	Test whether one pointer is greater than another
`>=_ptr_log`	... greater than or equal to ...
`<_ptr_log`	... less than ...
`<=_ptr_log`	... less than or equal to ...
`=_{float,double}_log`	Test whether two {floats, doubles} are equal
`!=_{float,double}_log`	... not equal
`>_{float,double}_log`	Test whether one {float, double} is greater than another
`>=_{float,double}_log`	... greater than or equal to ...
`<_{float,double}_log`	... less than ...
`<=_{float,double}_log`	... less than or equal to ...