The programming language raz

1. A first look
2. Words and symbols
3. Modules
4. Declarations
5. Types
6. Expressions
7. Statements
8. Procedures
9. Calling C
10. Programs
11. Versions
Appendix A. What an implementation chooses
Appendix B. The grammar

raz is a small systems language, first made for x86-64 Linux. A raz program can lay a record over a device's registers, read and write a device's memory exactly as written, take part in lock-free algorithms, make system calls and call C libraries. It has no garbage collector and no hidden run time: what the program does is what its text says.

This is the first volume of the report: the language, what an implementation chooses, and the grammar. The second, `doc/raz-lib.md`, is the library, which sections 3 and 10 and appendix A name.

This report defines the language. It is short on purpose. Whatever it doesn't allow, raz doesn't have.

This is version 2026-10.31 of the report; section 11 says where each version's changes are told. 2026-10.11 and 2026-10.12 are a thaw the owner chose, for a language anyone can build again. They break programs written for 2026-10.10 and before: for 2026-10.9, say, the report of compiler 0.5.0, the last release before them; and 2026-10.12 breaks code procedures written for 2026-10.11. 2026-10.20 is a third, for `try` (section 7.8); it breaks a program that has a name `try`. 2026-10.25 is a fourth, for strings: raw strings, which may run over lines and may be indented with the code around them (section 2), and `match` on a string (section 7.4); it breaks no program. 2026-10.26 is a fifth: an `INT` widens to an `I64` wherever it goes to one, in an expression too (sections 5.10 and 6.1); and a view, `v.(T)`, reads and writes the bytes of a variable as a `T` (section 6.9); it breaks no program. 2026-10.27 mends a defect: returning a slice of the procedure's own frame is an error (section 7.3), which breaks only a program that read memory already gone. 2026-10.28 mends another: an array a call returns lives to the end of its statement, and a slice of it kept past that is undefined (sections 8 and 10). 2026-10.29 is a sixth thaw: an array constant typed `[]T` takes its length from its literal (section 4); it breaks no program. 2026-10.30 mends a defect: `[]` used as a slice is the empty slice, which may be returned (sections 6.5 and 7.3); it breaks no program. 2026-10.31 is a seventh thaw: `INT` is at least 16 bits, and `U8`, `I8` and `I16` widen to it (sections 5.1, 5.10 and 6.1); an enumeration's name may be written with its type, and must be where another declaration shares it (section 5.7); and `continue` (section 7.2), which breaks a program that has a name `continue`.

The language is frozen, softly, from this version on: it changes only to mend a defect, and its library grows only by what breaks no program written for it. What anyone wishes it had waits on a list, for a later thaw.

1. A first look

```
module Hello;
import Out;

proc main*;
begin
    Out.str("hello, world\n");
end;

end Hello.
```

A program is a set of modules. The one named on the command line is the main module, and it exports a procedure `main`; the program starts there, once every module's body has run (section 3). Saved as `Hello.raz`, it is compiled with `raz Hello.raz`, which writes the executable `Hello`.

2. Words and symbols

A program is written in UTF-8. The compiler rejects a file that isn't valid UTF-8: a malformed or overlong sequence, a surrogate, or a code point beyond 10FFFFH, and a file with a NUL byte. Everything outside strings, characters and comments is ASCII. Spaces, tabs and line breaks separate symbols and are otherwise ignored.

Identifiers are an ASCII letter or _ followed by letters, digits and _, up to 63 characters. Case matters: `count` and `Count` are different names. _ alone isn't a name: it is the blank, which stands only to the left of `:=` (section 7.1).

Reserved words are lowercase, except `VOLATILE`:

```
and begin break by code const continue defer div do else elsif end
extern for forward if import in link match mod module not of or packed
proc record repeat return set shl shr then to try type until var
variant VOLATILE while xor
```

`static`, in `link static;` (section 3), and `check` and `off`, in `check off;` (section 3), aren't reserved.

Predeclared names are the basic types (section 5.1), `ERROR`, `true`, `false`, `nil` and the built-in procedures (section 6.8). They belong to a scope around every module, so a module or a procedure may declare the same name for something else, which hides them there.

Numbers. `123` is decimal. `0FFH` is hexadecimal: it starts with a digit, its digits are `0` to `9` and `A` to `F`, and it ends in `H`, all in capitals. `1.5` and `2.0E-3` are reals; a real always has a digit on both sides of its point, and its exponent, if it has one, is `E`, a sign if wanted, and digits. A decimal number above the largest `I64`, or a hexadecimal one above the largest `U64`, is an error; so is a real above the largest `F64`. A real below the least `F64` is 0.

An integer is an untyped constant, which takes the type it is used as, up to the largest `I64`. A hexadecimal number above that, such as `0FFFFFFFFFFFFFFFH`, has type `U64`: so `-1` never fits a `U64`, and the big number goes nowhere else without a conversion. Arithmetic on untyped constants doesn't wrap: a result beyond the `I64`s, such as `9223372036854775807 + 1`, is an error.

A real is an untyped constant too. It is read as the nearest `F64` (a tie goes to the even one), and takes `F32` or `F64`, whichever it is used as: an `F32` is the nearest to that `F64`. Where nothing says which, it is an `F64`. A real never becomes an integer, nor an integer a real, without a conversion: `x := 1;` is an error for `x: F64`, and `x := 1.0;` is right.

Characters. `'a'` is the code point of one character, an integer: `'a'` is 97 and `'é'` is 233.

Strings. `"héllo\n"` is a sequence of bytes, the UTF-8 of its text: `"é"` is the two bytes 0C3H, 0A9H. Both characters and strings take the escapes `\n` (line feed), `\t` (tab), `\r` (carriage return), `\0` (zero), `\\`, `\'`, `\"`, `\u{H...}` (the code point with hexadecimal value H…, up to 10FFFFH, in a string its UTF-8) and, in strings only, `\xHH` (the byte with hexadecimal value HH, which needn't make valid UTF-8). In an escape, the hexadecimal digits may be capitals or not. A character, or a string in double quotes, ends on the line it starts on.

Raw strings. `` `C:\tmp\"a b"` `` is a string written between backquotes: its bytes are the text between them as it stands, with no escapes, so `\` and `"` are themselves. A raw string may run over lines, and each line break in it is

a line feed, part of the string; a carriage return in it is left out.

A raw string whose opening backquote ends its line, and whose closing backquote has only blanks (spaces and tabs) before it on its line, is indented: the text is its lines between those two, each without the closing line's blanks at its start, and each ending in a line feed. So a long text may stand at the indentation of the code around it:

```
proc usage;
begin
    Out.str(`
        usage: tool [-v] file ...
          -v  say what is done
        `);
end;
```

writes `"usage: tool [-v] file ...\n -v say what is done\n"`. Each line of an indented raw string must start with the closing line's blanks, the same spaces and tabs in the same order, or it is an error; a line holding only blanks is an empty line of the text. A raw string that isn't indented is the text between its backquotes as it stands.

A raw string can't hold a backquote; a text that needs one joins it with +, as `"`"`. Everywhere else a raw string is a string, as one in double quotes is.

Messages give the line and the column, counted in characters (section 10).

Comments are `(* … *)` and nest. By convention, the comment just before an exported declaration is its documentation, and the one before `module` is the module's.

Operators and delimiters:

```
+  -  *  /  &  |  ~  =  #  <  <=  >  >=  :=  :  ;  ,  .  ..  ...  (  )  [  ]  {  }  ^
```

`|` is also what starts an arm of `match` (section 7.4); an arm comes where a statement could, never after an operand, so the two don't mix.

3. Modules

```
Module     = "module" ident [ "(" ModParams ")" ] ";"
             { Import } { Link } [ Check ] { Declaration }
             [ "begin" Statements ] "end" ident "." .
ModParams  = ident { "," ident } { ";" ident ":" "proc" Signature } .
Import     = "import" ImportItem { "," ImportItem } ";" .
ImportItem = ident [ ":=" ident [ "(" Type { "," Type } ")" ] ] .
Link       = "link" ( string | "static" ) ";" .
Check      = "check" "off" ";" .
```

A module lives in a file of its own name: module `Out` is `Out.raz`. The name after `end` repeats the module's name.

Exports. A name declared at the top of a module with `*` after it is exported: other modules can use it as `Module.name`. Record fields are exported the same way. Everything else is private to the module. An exported variable can be read by the modules that import it, but only its own module changes it. Outside it, the variable is a value, not a variable: nothing that could write it, or let something else write it, may name it or a field or element of it. So it can't be assigned, passed as a `var` argument, changed by `INC`, `DEC` or `NEW`, or given to `ADR`; an array in it can't be sliced, or used where a slice is expected (the silent conversion of an array, section 5.10), though a slice in it may be sliced, since that makes a new slice and can't change the variable; in `match`, its arms' names stand for the fields of a copy. It can be read, copied and compared, and a copy is the importer's own. What a pointer or a slice in it points at is another variable, and isn't held to this: with `s: []U8` exported, `M.s[0] := 0` writes what `M.s` points at, not `M`. A module that means to share an array exports a pointer to it, or a procedure that returns a slice of it.

Imports. `import Out;` makes `Out`'s exports available as `Out.x`. `import O := Out;` uses the name `O` instead. A module can't import itself, directly or through others.

Generic modules take types as parameters:

```
module List(T);
type
    Node* = record next*: ^Node; val*: T; end;
...
end List.
```

An import gives the arguments: `import Ints := List(I64);`. Each distinct list of arguments makes one *instance*: the module compiled with `T` standing for `I64`. Two imports of `List(I64)`, in any modules, share one instance, its variables and its `ERROR`s with it: two lists of arguments are the same when their types are the same (section 5.10), an alias as what it stands for, and their procedures are the same procedures. Since every instance is checked on its own, an operation the argument type doesn't support (adding two records, say) is reported as an error in that instance.

An instance is made where its module is first used, not at the import: so the arguments may name types the importing module declares after its imports, as long as they are declared before that first use. The instance is made before the code of the procedure, or the module body, that first uses it, and its arguments are read in the scope there.

```
import Table := Map([]U8, Colour, Str.hash, Str.eq);

type Colour = (red, green, blue);

var names: Table.Map;   (* the instance is made here *)
```

What a generic module needs to do with its types, beyond moving them about, it is given as procedures, after the types:

```
module Map(K, V; hash: proc(k: K): U64; eq: proc(a, b: K): BOOL);
```

```
import Names := Map([]U8, I64, Str.hash, Str.eq);
```

A procedure argument names a procedure, not a variable or an expression, and its signature, with the type arguments put in, must be the parameter's. Inside the module, `hash(k)` calls it directly. Two imports make the same instance only if their types and their procedures are the same.

Finding modules. The compiler looks for module `Name` in the main module's directory, then in each directory given with `-I`, then in the `lib` directory beside the compiler. In each directory it takes the first of `Name.OS-ARCH.raz`, `Name.OS.raz`, `Name.ARCH.raz` and `Name.raz` that exists, where `OS-ARCH` is the target, for now only `linux-x86_64`. So a module whose code depends on the machine or the system comes in one file per target, and the modules that import it don't change. Machine code (section 8) and system calls belong in such files.

One module to a name. A program has one module of each name, so the compiler reports an error, naming the files, if a module it looks for has files in more than one of these directories, rather than take the first: a `Buf.raz` beside a program would otherwise stand in for the library's `Buf` in every module that imports it, the library's own among them. A link is the exception: a file that is a symbolic link to one of the module's files in a later directory chooses that file, so that a directory holding a link to `lib/Sys.raz` builds a program over the C library's `Sys` on Linux too.

Names are flat. The names of a program's modules are one space, with no packages or paths. So the names of the library's modules (the library, `doc/raz-lib.md`) are kept for it: a program or another library doesn't name a module of its own `Arena`, `Buf`, `C`, `Files`, `Fmt`, `Heap`, `List`, `Lock`, `Map`, `Math`, `Mem`, `Opt`, `Out`, `Rt`, `Sort`, `Str`, `Sys` or `Thread`, nor any name the library takes later. The modules of any other library share a prefix of their own, as an X11 library's `Xauth` and `Xkeys` do, and a program that wants a shorter name imports one: `import Auth := Xauth;`.

The compiler holds every file of such a module to the same interface. When it takes one of `Name.*.raz`, it also reads the declarations of the others beside it, and of `Name.raz` if there is one, and reports an error unless they all export the same names, each the same kind of thing:

- a constant of the same type (its value may differ: a system call number, say);
- a type of the same kind, and for a record or variant the same exported fields and cases, with the same types (unexported fields, and so the size and layout, may differ);
- a variable of the same type;
- a procedure of the same signature.

Two types here are the same if they are built alike, a name in one file standing for the same name in another.

Module bodies. A module may end with `begin` and statements, its body. Every module's body runs once, before the main module's `main`: a module's body runs after the bodies of the modules it imports, in the order of its `import`s, and once however many modules import it. Each instance of a generic module has its body run once. Module variables are zero before any body runs. `return;` ends a body early.

`link "libX11.so.6";` names a shared library to load at run time, for the `extern` declarations of section 9. **`link static;`** says the module needs a program without C (section 10): it makes its own system calls, or its own threads, which the C library wouldn't know about.

`check off;`, after the links, leaves out of the module's own code the checks of indexes and slice bounds of section 6.5, and that a slice of a pointer doesn't run backwards (section 6.4), for speed where the program is sure of them. An index out of range there reads or writes whatever memory it reaches, and what follows isn't

defined. Every other check stays, and the module's callers keep theirs. An instance of a generic module follows the generic module's own `check` line, not that of the module that made it.

4. Declarations

```
Declaration = "const" { IdentDef [ ":" Type ] "=" Expr ";" }
            | "type" { IdentDef "=" Type ";" }
            | "var" { IdentList ":" Type ";" }
            | ProcDeclaration ";"
            | ExternDeclaration ";" .
IdentDef    = ident [ "*" ] .
IdentList   = IdentDef { "," IdentDef } .
```

A name must be declared before it is used, with two exceptions: the base type of a pointer may name a type declared later in the same `type` section (so that two records can point at each other), and a procedure may be declared `forward` and defined later (section 8).

Constants are evaluated by the compiler. Their expressions use only literals, other constants, operators, conversions, `SIZE` and the `LEN` of arrays and strings. A constant is an integer, a real, a `BOOL`, a name of an enumeration, a set, a string, an array, a record or an `ERROR`. A constant of an integer type wraps as the program would: `U8(250) + 10` is 4. Dividing by zero in a constant is an error, and so is dividing the smallest value of a signed type by -1, with `div` or `mod`, which stops a program (section 6.1). A string constant can be indexed and sliced as an array of `U8` is, and its `LEN` is its length in bytes. `+` joins two string constants into one:

```
const Text = "the cat";
    Line = Text + "\n";
...
w := Text[4..7];    (* "cat" *)
```

A constant with a type is an `ERROR` (section 5.8), an array or a record, and the value of an array or record constant is a literal (section 6.5) of constants. Its elements and fields are integers, reals, `BOOL`s, enumerations, sets, `ERROR`s, procedures, `nil`, a string for a `[]U8`, an array constant for a slice of its element type, and arrays and records of these; no variant and no other pointer:

```
const Primes: [5]I64 = [2, 3, 5, 7, 11];
type Rule = record class: []U8; tag: INT; float: BOOL; end;
const Rules: [2]Rule = [{class: "firefox", tag: 2}, {class: "mpv", float: true}];
...
p := Primes[i];
```

It is used as a variable of that type is, but can't be changed: an array constant can be indexed and sliced and its `LEN` is `n`, a record constant's fields can be read, and either can be assigned, compared as its type allows, and passed. Its parts are constant as a string's bytes are (section 5.10). A part of it isn't a constant, even at a constant index.

An array constant may be typed `[]T`, with no length: its length is then its literal's, as a string constant's is the string's, and it is an array constant of that length, `[n]T`, in all else. Its literal has at least one element.

```
const Wide: [][2]I32 = [[1100H, 115FH], [2E80H, 303EH], [0AC00H, 0D7A3H]];
    Count = LEN(Wide);    (* 3, a constant *)
```

Variables of a module live as long as the program; variables of a procedure live as long as its call. Both start at zero: every byte of them is zero. So a number starts as `0` (`0.0` for `F32` and `F64`), a `BOOL` as `false`, a pointer, procedure or `ERROR` as `nil`, a slice as the empty slice (`ptr` is `nil`, `len` is 0), a variant as its first case with its fields zero, a set as the empty set, and an enumeration as its first name. No variable is ever read before it has a value.

5. Types

```
Type      = QualIdent
          | "[" Expr "]" Type                (* array *)
          | "[" "]" Type                     (* slice *)
          | "^" [ "VOLATILE" ] Type          (* pointer *)
          | [ "packed" ] "record" { IdentList ":" Type ";" } "end"
          | "variant" { Case ";" } "end"
          | "(" ident { "," ident } ")"      (* enumeration *)
          | "set" "of" QualIdent
          | "proc" Signature .
Case      = ident [ "(" FieldList { ";" FieldList } ")" ] .
FieldList = ident { "," ident } ":" Type .
QualIdent = [ ident "." ] ident .
```

5.1 Basic types

Type	Size	Values
`I8` `I16` `I32` `I64`	1, 2, 4, 8	signed integers, two's complement
`INT`	an address's: 8 on x86-64	signed, as big as an address
`UINT`	an address's	unsigned, as big as an address
`U8` `U16` `U32` `U64`	1, 2, 4, 8	unsigned integers
`BOOL`	1	`false` (0), `true` (1)
`F32`	4	IEEE 754 single precision
`F64`	8	IEEE 754 double precision

`INT` is the everyday integer: lengths, indexes and sizes are `INT`s. It is as big as an address, and at least 16 bits, as `UINT` is, on every target; so it is the integer a pointer converts to (section 6.7), and a word the atomic built-ins work on (section 6.8). `INT` and `I64` are two types, even where they have the same size: an `INT` goes silently into an `I64` wherever it meets one, in an expression too (sections 5.10 and 6.1), and an `I64` becomes an `INT` only by `INT(x)`, which may drop bits on a smaller target. A type bigger than the largest `INT` in bytes is an error.

`F32` and `F64` exist only on a target that has reals (appendix A). On one that hasn't, a real literal, and a use of `F32`, `F64` or of a name whose type is made of them, is an error; a declaration that no used code names isn't, so a module like `Fmt` keeps one interface on every target.

Each is aligned as the target's C aligns it: to its size on x86-64. So are `ERROR` (section 5.8), pointers and procedures, each as big as an address; a slice is aligned as an address is.

5.2 Arrays

`[n]T` is `n` elements of type `T`, one after another. `n` is a constant greater than zero. Arrays are values: assigning one copies it.

5.3 Slices

`[]T` is a view of elements that live elsewhere: a pointer to the first and a count, two addresses' worth, 16 bytes on x86-64.

```
Slice = record ptr: ^T; len: INT; end;    (* its layout *)
```

A slice is made from an array (`a[2..5]`, or an array used where a slice is wanted), from another slice, or from a pointer (`p[0..n]`). It doesn't own what it points at.

5.4 Pointers

`^T` is the address of a `T`, as big as an `INT`. `nil` is the pointer to nothing.

`^VOLATILE T` is the address of a `T` that something besides the program may read or change: a device register, memory shared with another process. Section 7.6 says what that promises.

5.5 Records

```
type Cell = record ch: U8; attr: U8; end;
```

Fields are laid out in order as C lays them out: each at the next multiple of its own alignment, and the whole record padded to a multiple of its largest alignment. A record declared like a C struct has the same layout as that struct. A record may have no fields; its size is then 0.

A `packed record` has no padding: each field starts at the byte after the field before it, and the record's alignment is 1. So a field of a packed record may sit at any address. It is read and written like any other, on every target, even one whose machine can't reach a misaligned word in one access: the compiled code reaches it in parts there. But a field of a packed record whose type is aligned to more than 1 byte, and anything inside it, has no address: `ADR` of it, a `var` parameter of it and a slice of it are errors. Nor can it be reached through a `^VOLATILE` pointer, since a misaligned access can't be promised as one access (section 7.6). Its value can be copied out and back in.

The bytes of a value of more than one byte are in the target's order (appendix A). A file or a message in an order of its own is read and written a byte at a time, as `Str.le`, `Str.be`, `Str.putLE` and `Str.putBE` do (the library, `doc/raz-lib.md`).

5.6 Variants

A variant holds one of several cases, each with its own fields:

```
type
    Shape* = variant
        circle(r: F64);
        rect(w, h: F64);
        dot;
    end;
```

The case names are declared in the scope around the type, so `circle` can be used directly, and as `Shapes.circle` from another module if `Shape` is exported. A case with fields is built by calling it like a procedure, `circle(2.0)`; a case without fields is just `dot`. The only way to read the fields is `match` (section 7.4).

Layout: an `INT` tag (0 for the first case, 1 for the second, and so on), then the space for the fields. Each case's fields are laid out in that space as a record's would be, after the tag, and the cases share it, so the variant is as big as its largest case needs, padded to its alignment, the largest of an `INT`'s and its fields'. `variant a(x: U8); b(y: I64; z: U8); end` is 24 bytes on x86-64. A variant that starts at zero therefore holds its first case.

5.7 Enumerations and sets

```
type
    Color* = (red, green, blue);
    Colors* = set of Color;
```

An enumeration is a list of names, in order. Like the cases of a variant, the names are declared in the scope around the type. An enumeration has at most 256 names and takes 1 byte; the first name is 0, the next 1, and so on.

A name of an enumeration may also be written with its type, `Color.red`, and one imported with the type's module, `M.Color.red`. Two enumerations of a scope may share a name, and an enumeration may have a name that something else of its scope has, a constant, a variable or a procedure: such a name is written with its type, `Elem.bold`. Written alone, it names the one declaration of it that isn't an enumeration's, if there is one, and is an error if there isn't:

```
type
    Style = (bold, italic);
    Elem = (para, bold, rule);
proc rule;                  (* a procedure, beside Elem's rule *)
...
s := Style.bold;  e := Elem.bold;  e := Elem.rule;
e := para;                  (* para is Elem's alone *)
rule;                       (* the procedure *)
s := bold;                  (* an error: write Style.bold or Elem.bold *)
```

The type is never taken from where the name stands: `s := bold` is an error though `s` is a `Style`.

`set of E`, for an enumeration `E` of at most 64 names, holds any choice of `E`'s names: bit `i` stands for the name at position `i`. It takes the smallest of 1, 2, 4 and 8 bytes that has a bit for each name, so a set can stand for a C flags word.

5.8 Errors

```
const NotFound*: ERROR = "not found";
```

An `ERROR` is how a procedure says it failed: by convention the last of its results, and `nil` when nothing failed. A caller that can't deal with a failure itself passes it on with `try` (section 7.8).

`ERROR` is a basic type as big as an address. A value of it is made only by a constant declared as above, and each such declaration makes an error different from every other, whatever its message: two modules that both declare `"not found"` have two different errors. A generic module makes its errors once per instance. `nil` is no error.

Errors compare with `=` and `#` only, by which declaration made them, never by their message. `MSG(e)` is the message, a `[]U8`; `MSG(nil)` is `""`.

```
proc find*(key: I64): (^Node, ERROR);
...
n, err := Table.find(7);
if err = Table.NotFound then ... end;
```

5.9 Procedure types

```
type Compare = proc(a, b: I64): BOOL;
```

A value of a procedure type is the address of a procedure with that signature, as big as an address, or `nil`. Two signatures are alike when their parameters have the same types, each `var` or not alike, in order, and their results are the same types; the parameters' names don't matter.

5.10 Same and assignable

Two named types are the same only if they are the same declaration. Two anonymous pointer, array, slice, set or procedure types are the same if they are built alike: `^I64` is `^I64` wherever it is written. A record, variant or enumeration, though, is a type of its own wherever it is written, named or not: two `record x: I64; end` are two types. A named and an anonymous type are the same if the named one is built like the anonymous one, so that `p := ADR(n)` works for `p: NodePtr` with `type NodePtr = ^Node`. Sameness doesn't pass through the anonymous type: with `type P1 = ^Node; P2 = ^Node;`, each of `P1` and `P2` is the same as `^Node`, but they are two declarations, so not the same as each other.

A type declared with only a name on the right, `type Count = I64;`, is another name for that type, not a new one.

A value of type `A` can be assigned to a variable of type `B` (and passed to a parameter of type `B`, and returned as a `B`) if they are the same type, or if:

- `A` is an integer constant whose value fits in the integer type `B`;
- `A` is `nil` and `B` is a pointer, procedure or `ERROR` type;
- `A` is `^T` and `B` is `^VOLATILE T`;
- `A` is `[n]T` and `B` is `[]T`;
- `A` is a string literal and `B` is `[]U8`, `^U8` (a pointer to the bytes followed by a zero byte, as C wants), or `[n]U8` with `n` at least the string's length (the rest is zero);

- `A` is a procedure and `B` a procedure type with the same signature;
- `A` is `INT` and `B` is `I64`, or `A` is `UINT` and `B` is `U64`, which loses nothing on any target;
- `A` is `U8`, `I8` or `I16` and `B` is `INT`, or `A` is `U8` or `U16` and `B` is `UINT`, which an `INT` or a `UINT` of 16 bits holds whole, so on every target; unless the value is what an operator gives (below).

A string's bytes are constant. A `[]U8` or `^U8` made from a string literal, or a slice of a string constant, points at them, and writing through it isn't defined: on a target with read-only memory, it may stop the program as a bad address (section 10). A `[n]U8` made from a string is a copy, the program's to change.

Nothing else converts silently. In particular, there is no other conversion between integer types, between integers and reals, or between `F32` and `F64`, without a written conversion (section 6.7). An `INT` widens to `I64`, and a `UINT` to `U64`, wherever its value goes where an `I64` is wanted: in `a, b := f(x)` and `return f(x)` as in any other assignment, to the bounds of a `for` and the step of `INC` and `DEC` whose variable is an `I64`, and in an expression, where it meets an `I64` (section 6.1). The types themselves stay two: a `^INT` isn't a `^I64`, an `INT` variable isn't passed to a `var` parameter of type `I64`, and a procedure returning an `INT` isn't of a procedure type returning an `I64`.

A `U8`, `I8` or `I16` widens to `INT`, and a `U8` or `U16` to `UINT`, in the same places, as an `INT` to `I64`: assigned, passed, returned, in `a, b := f(x)`, to a `for`'s bounds and `INC` and `DEC`'s step, and in an expression, where it meets an `INT` or a `UINT` (section 6.1); not in `return f(x)`, which passes `f`'s results on as they are. They widen only that far: a `U8` meets an `I64` only as `INT(b)` or `I64(b)`. And they widen only as values that are moved: a variable, an element, a field, a call's result or a conversion's. What an operator gives on them doesn't: it was computed in their type, and has wrapped there. With `a, b: U8` holding 200 and 100, `a + b` is 44, so `n := a + b`, for an `INT` `n`, is an error; `n := INT(a) + b` is 300, and says where the sum is made.

```
var s: []U8; n, d: INT;
n := s[i];                  (* an element: widens *)
n := n * 10 + s[i] - '0';   (* n * 10 + s[i] is an INT, and so the rest *)
d := s[i] - '0';            (* an error: a U8, which may have wrapped *)
d := INT(s[i]) - '0';       (* below '0' too, it is negative *)
```

6. Expressions

```
Expr       = Conjunct { "or" Conjunct } .
Conjunct   = Relation { "and" Relation } .
Relation   = SimpleExpr [ RelOp SimpleExpr ] .
RelOp      = "=" | "#" | "<" | "<=" | ">" | ">=" | "in" .
SimpleExpr = [ "-" ] Term { AddOp Term } .
AddOp      = "+" | "-" | "|" | "xor" .
Term       = Factor { MulOp Factor } .
MulOp      = "*" | "/" | "div" | "mod" | "&" | "shl" | "shr" .
Factor     = number | char | string | Designator | "(" Expr ")"
           | "not" Factor | "~" Factor
           | "[" [ Expr { "," Expr } ] "]"
           | "{" [ Element { "," Element } ] "}"
           | "^" [ "VOLATILE" ] Type "(" Expr ")"
           | "try" Designator .
Element    = Expr | ident ":" Expr .
Designator = QualIdent { Selector } .
Selector   = "." ident | "^" | "[" Expr [ ".." Expr ] "]"
           | "(" [ Expr { "," Expr } ] ")" | "." "(" Type ")" .
```

The operators bind in the order of the grammar: `not`, ~ and `try` (section 7.8) most tightly, then the multiplying operators, then the adding ones, then the relations, then `and`, then `or`. So `a + b * c = d` is `(a + (b * c)) = d`, `x & 0FH = 0` is `(x & 0FH) = 0`, and `i < n and a[i] # 0 or done` is `((i < n) and (a[i] # 0)) or done`. A unary minus applies to the first term: `-a * b` is `-(a * b)`.

`and`, `or` and `not` are for `BOOL`s only; `&`, `|`, `xor` and ~ are for integers only.

A pointer to a record isn't the record: its fields are `p^.f`, never `p.f`, and the elements of a pointer to an array are `p^[i]`. A statement starts with a name or `try`, so `^T(x)^ := v` isn't one; a view, `mem[a].(I64) := v` (section 6.9), says it without a pointer.

6.1 Integers

Both operands of `+ - * div mod & | xor`, and of a relation, have the same integer type, or one is an untyped constant that takes the other's type, which it must fit, or one is an `INT` and the other an `I64` (a `UINT` and a `U64`), and the `INT` widens to `I64`, or one is an `INT` and the other a `U8`, `I8` or `I16` (a `UINT` and a `U8` or `U16`) that no operator gave, and it widens to `INT` (section 5.10). So do `v` and `k` in `INC(v, k)` and `DEC(v, k)`, where only `k` widens: an `I64` `k` isn't added to an `INT` `v`. The result has that type and wraps around at its width: in `U8`, `200 + 100` is `44`. A unary minus wraps too: for `x` of an unsigned type, `-x` is `0 - x`. Other integer types are never mixed, whatever their sizes: with `i: I32`, `i < LEN(s)` is an error, and `i` should be an `INT`; with `c: U8`, `c < LEN(s)` is an `INT` comparison. With `pos: I64`, `pos + LEN(s)` is an `I64`.

Where the report asks for an integer and names no type, as an index, a slice's bound, a shift's count, a pointer's offset `i` in `p + i`, or a conversion's operand, any integer type will do, and its value is taken. On a target whose `INT` is narrower, a pointer's offset is cut to an `INT`.

`SIZE(T)`, and `LEN` of an array or a string constant, are untyped constants, like a literal.

`div` rounds toward zero and `mod` takes the sign of the dividend, as in C: `-7 div 2` is `-3`, `-7 mod 2` is `-1`. Dividing by zero, and dividing the smallest value of a signed type by `-1`, stop the program (section 10),

whatever the type and the target.

`&` (and), `|` (or), `xor` and `~` (not) work on every bit. `x shl n` and `x shr n` shift by `n` modulo the width of `x`'s type, in bits: for `x: U8`, `x shl 9` is `x shl 1`. `shr` brings in copies of the sign bit for signed types and zeros for unsigned ones. `n` may be of any integer type. The result has `x`'s type, cut to its width: in `U8`, `0C0H shl 1` is `80H`. If `x` is an untyped constant, so is the result, and it doesn't wrap: `1 shl 9` is 512, which doesn't fit a `U8`.

Relations compare as signed or unsigned according to the type.

6.2 Booleans

`and` and `or` evaluate their right operand only when it decides the result. `not` negates, and `=` and `#` compare; `#` is "different".

6.3 Reals

`+ - * /` and the relations work on `F32` and on `F64`. Both operands have the same type, or one is a real constant, which takes the other's type (section 2). The result has that type: `F32` arithmetic is done in single precision, rounded at each step as C rounds `float`. A constant expression of reals is evaluated in double precision, then rounded once to the type it takes.

Relations are IEEE 754's: a NaN is equal to nothing, itself included, so for a NaN `x`, `x # x` is true and every other relation with it is false. Dividing by zero gives an infinity or a NaN; arithmetic on reals never stops the program.

6.4 Pointers

- `p^` is the variable `p` points at.
- `p + i` and `p - i`, with `i` an integer, move `p` by `i` elements: by `i * SIZE(T)` bytes for `p: ^T`.
- `p - q`, with both of the same pointer type, is the number of elements between them, an `INT`.
- `p[i]` is `(p + i)^`. `p[lo..hi]` is the slice of the `hi - lo` elements from `p + lo`. Neither is checked against a length, which a pointer doesn't have, but a slice of a pointer that runs backwards, `hi < lo`, is a failed check (section 6.5). A pointer to an array isn't indexed this way, since a step would be a whole array: its elements are `p^[i]`.
- Pointers compare with `= # < <= > >=` as unsigned addresses. Procedure values compare with `=` and `#` only.

6.5 Arrays and slices

`a[i]` is an element of an array or slice, and `a[lo..hi]` is the slice of the elements from `lo` up to but not including `hi`. Both are checked: `i` must be below the length, and `0 <= lo <= hi <= length`. A check that fails prints to standard error where it is, as file, line and column, and what failed, with the values involved (`index 7 is out of range for a length of 5`), and stops the program with status 2. An index or a bound may be of any integer type, and is checked by its value, whatever its type: a negative `I8`, or a `U64` past the length, fails like any other. A module that says `check off;` (section 3) leaves these checks out of its code.

Literals. `[e1, e2, ...]` builds an array, and `{f: e, g: e, ...}` a record, where the type it must have is known: assigned to a variable, passed to a parameter, returned, compared with a value of that type, or an element of a bigger literal. The elements must be assignable to the element type, and an array literal for `[n]T` has exactly `n` of them. A record literal names its fields, in any order, each at most once; the fields it leaves out

are zero, so `{}` is the record with every field zero. An array literal used as a slice is a hidden variable of the procedure, and lives as long as its call; the empty one, `[]`, is the empty slice, as a slice variable starts (section 4), and no variable at all.

```
c := {ch: 'A', attr: 7};
Fmt.print("{} of {}\n", [Fmt.int(n), Fmt.int(m)]);
```

Arrays and strings compare. Two operands that are each a `[]T` or a `[n]T`, of the same element type `T`, are `=` when they have the same length and the same elements, and `#` otherwise, where `T` is an integer type, `BOOL`, an enumeration, a set or a pointer type. A string literal compares as a `[]U8`. The comparison is code of its own where it stands, a check of the lengths and then of the bytes, not a call.

Arrays of reals don't compare this way, since `-0.0 = 0.0` and a NaN is equal to nothing, though their bytes say otherwise; nor do arrays of records, variants or procedures, nor records and variants themselves. Write a loop. Ordering strings is the library's `Str.cmp`.

6.6 Enumerations and sets

Enumerations compare with all six relations, by position, and an enumeration can be the variable of a `for` loop and the expression of a `match` (section 7.4).

`{red, blue}` is a set, of the set type of its names' enumeration; `{}` is the empty set, where the type it must have is known. On two sets of the same type, `+` is union, `*` intersection and `-` difference; `=` and `#` compare, and `<=` and `>=` are "is a subset of" and "is a superset of". `x in s` is `true` if `x`, a value of the enumeration, is in the set `s`.

6.7 Conversions

A type name applied like a procedure converts: `I32(x)`, `F64(n)`, `^Cell(addr)`. A pointer type written out uses the `^` form of the grammar: `^VOLATILE U32(0FEE00000H)`.

From	To	Result
integer	integer	the value cut to the new width, or extended by its sign (signed) or zeros (unsigned)
integer	`F32`, `F64`	the nearest real
`F32`, `F64`	integer	rounded toward zero; a NaN, or a real out of the range of `I64`, gives the smallest `I64`, then cut
`F32`	`F64`	the same value
`F64`	`F32`	the nearest `F32`; too big a value gives an infinity
enumeration	integer	its position
integer	enumeration	the name at that position; a position out of range is a failed check (6.5), and in a constant an error
set	integer	its bits
integer	set	the low bits, as many as the set has

From	To	Result
`INT`, `UINT`	pointer, procedure	the address
pointer, procedure	`INT`, `UINT`	the address
pointer, procedure	pointer, procedure	the same address, seen as the new type

There are no others: no conversion between `BOOL` and integers, none between pointers and integers other than `INT` and `UINT`, and none to or from records, arrays, slices, variants or `ERROR`. An untyped integer constant converts to a pointer as an `INT` does, as in `^VOLATILE U32(0FEE00000H)`; it isn't assigned to a pointer without the conversion. A conversion of a constant is made by the compiler, by the same rules: `U8(300)` is 44.

6.8 Built-in procedures

Call	Result
`SIZE(T)`	the size of type `T` in bytes, an untyped constant; `T` is a type, not a variable
`ADR(x)`	the address of variable `x`, a `^T`
`LEN(a)`	the number of elements of an array or a string constant (an untyped constant) or of a slice (an `INT`)
`NEW(p, a)`	sets `p: ^T` to a fresh, zeroed `T` from the allocator `a: ^Mem.Allocator`, or to `nil` if `a` has no room
`MSG(e)`	the message of the error `e`, a `[]U8`
`COPY(d, s)`	copies the elements of `s` to the start of `d`, both arrays or slices of one type `T`; `LEN(s)` must be at most `LEN(d)`
`SYSCALL(n, x1, ..., x6)`	Linux system call `n` with up to six arguments, each a pointer or an integer no wider than `INT`; the raw result, an `INT` (a negative number is `-errno`)
`INC(v)`, `INC(v, k)`	adds 1, or `k`, to the integer variable `v`; it wraps, like `+`
`DEC(v)`, `DEC(v, k)`	subtracts 1, or `k`, from `v`
`FENCE`	a full memory barrier
`CAS(p, old, new)`	if `p^ = old`, sets `p^ := new` and returns `true`; atomically
`XCHG(p, x)`	sets `p^ := x` and returns the value before, atomically
`XADD(p, x)`	adds `x` to `p^` and returns the value before, atomically
`ARGC()`	the number of command line arguments, an `INT`
`ARGV()`	the arguments, a `^^U8` (C strings)

`SYSCALL` exists only on Linux targets. Other systems have no stable system call interface, and their modules call the C library instead (section 9).

`NEW(p, a)` calls `a^.alloc(a, SIZE(T), A)`, where `A` is the alignment of `T`, and zeroes the bytes it gets if they aren't `nil`. `Mem.Allocator` (the library, `doc/raz-lib.md`) is the one type of the library the language names.

`COPY` is a statement, as `INC` is. It copies the bytes of `LEN(s)` elements as one block, as an assignment of an array does, and is right when `d` and `s` overlap, as in `COPY(a[1..n], a[0..n - 1])`. It is checked as a slice is: past `LEN(d)`, it stops the program, unless the module says `check off;`.

`INC` and `DEC` evaluate `v` once, so `INC(a[f(i)])` calls `f` once. They read `v` and then write it: two accesses, even through a `^VOLATILE` pointer, and not atomic; `XADD` is the atomic one.

For `CAS`, `XCHG` and `XADD`, `p` is a `^T` or `^VOLATILE T` where `T` is `INT`, `UINT` or a pointer type (`INT` or `UINT` for `XADD`), on every target. They are sequentially consistent, on every target; how a machine without atomic instructions makes them so is the implementation's choice (appendix A). ### 6.9 Views

A view reads and writes the bytes of a variable as a value of another type. `v.(T)`, with `v` a variable, is the `T` whose bytes start where `v` starts:

```
var mem: [4096]U8; n: I64; x: F64; bits: U64;

mem[16].(I64) := -1;     (* the 8 bytes mem[16] to mem[23] now hold -1 *)
n := mem[16].(I64);      (* and are read back as an I64: n is -1 *)
bits := x.(U64);         (* the 8 bytes of x, as a U64 *)
```

A view is a variable, not a value: `mem[16].(I64)` is made of the 8 bytes `mem[16]` to `mem[23]`, as `p^` is the variable `p` points at. Reading it reads those bytes; assigning it writes them, and so changes `mem`. Nothing is copied to make it. It stands wherever a variable may: on the left of `:=`, as the variable of `INC`, and with selectors after it:

```
type Header = record magic: U32; count: U32; end;
var buf: [512]U8; k: U32;

k := buf[0].(Header).count;    (* bytes 4 to 7 of buf, as a U32 *)
buf[0].(Header).count := 0;
INC(mem[a].(I64), 8);
```

A view isn't a conversion. A conversion works on a value, a view on bytes. With `x: F64` holding 1.5, `I64(x)` is 1, the number rounded toward zero, while `x.(I64)` is 3FF8000000000000H, the bits that hold 1.5. With `n: I64` holding 258, `U8(n)` is 2, the number cut to its low byte, while `n.(U8)` is the first byte of `n` in memory, which is the low one only on a target that puts its low byte first (appendix A). When the program means a number, it wants a conversion; when it means the bytes as they lie, a view.

The types. `T` is an integer type, a real type, a pointer type, or a record or array made of these alone, at any depth: the types for which any bytes at all are a value. `BOOL`, an enumeration, a set, a slice, a variant, a procedure type and `ERROR` are left out, since bytes that are none of their values would make the program mean nothing. To read one of them from bytes, view the integer and convert it: `Colour(mem[a]).v` may be of any type, and is a variable: a name of one, a field or element of one, or what a pointer points at; not a constant, nor the result of a call or of an operator.

Where its bytes lie. The `T` lies within the variable `v` is part of:

- for an element of an array or a slice, `a[i].(T)`, from that element to the end of `a`: `mem[4088].(I64)` is the last 8 bytes of `mem`, and `mem[4090].(I64)` would run 2 bytes past its end;
- for anything else, within `v` itself: `x.(U64)` is allowed only if `x` has at least 8 bytes.

This is checked, as an index is: by the compiler when it can tell, otherwise while the program runs, where a view that runs past the end is a failed check (section 6.5), unless the module says `check off;`. An element reached through a pointer, `p[i].(T)`, is checked no more than `p[i]` is, since a pointer has no length.

Any address. The `T` may start at any byte: `mem[3].(I64)` is an `I64` whose first byte is `mem[3]`. As a field of a packed record is (section 5.5), it is read and written on every target, in parts where the machine needs that; and, as such a field, a view has no address: `ADR` of it, a `var` argument of it and a slice of it are errors. Copy it out to a variable, and back in. A view of a volatile variable is an error too, since its access can't be promised as one (section 7.6): a device's registers are reached through a `^VOLATILE` pointer to a record of them.

Byte order. The bytes are in the target's order (appendix A). A file or a message whose format fixes an order of its own is read with `Str.le`, `Str.be` and the like (section 5.5), if the program is to read it right on every target.

Before views. A program used to make a pointer for each access, and may still: `mem[a].(I64) := n` does what

```
p := ^I64(ADR(mem[a]));
p^ := n;
```

does, but with `a` checked against the length of `mem`, no pointer variable, and no address of a word that may be misaligned.

`v` is evaluated once, and its check made, before the bytes are read or written.

7. Statements

```
Statements = { Statement ";" } .
Statement  = Designator [ ":=" Expr ]
           | "_" ":=" Expr
           | Target "," Target { "," Target } ":=" Expr
           | "if" Expr "then" Statements
             { "elsif" Expr "then" Statements } [ "else" Statements ] "end"
           | "while" Expr "do" Statements "end"
           | "repeat" Statements "until" Expr
           | "for" ident ":=" Expr "to" Expr [ "by" Expr ] "do" Statements "end"
           | "match" Expr "of" { "|" Arm } "end"
           | "return" [ Expr { "," Expr } ]
           | "break"
           | "continue"
           | "defer" Statement
           | "try" Designator .
Arm        = ( ident [ "(" ident { "," ident } ")" ]
             | Label { "," Label } | "else" ) ":" Statements .
Label      = Expr [ ".." Expr ] .
Target     = Designator | "_" .
```

Every statement ends in `;`, including the ones that end in `end` or `until`.

7.1 Assignment and calls

`x := e` evaluates `e` and stores it in `x`; arrays, records, slices and variants are copied. A designator on its own is a call, and any result it returns is dropped: `Out.ln;`, `write(fd, buf);`.

`a, b := f(x);` assigns the results of a procedure that returns several (section 8), in order.

The blank, _, drops what would be assigned to it: `_ := e;` evaluates `e` and drops it, and `n, _ := f(x);` keeps the first result and drops the second. What is dropped is still evaluated as any expression is: its calls are made, and a volatile variable in it is read (section 7.6). A result dropped by a call standing on its own is a lint an implementation may warn of (section 10); one dropped through _ says it is meant.

7.2 Conditions and loops

`if`, `while` and `repeat` take `BOOL` expressions. `repeat` runs its body at least once and stops when the condition is true.

`for i := a to b by s do ... end;` sets `i` to `a`, `a + s`, … while `i <= b` (or `i >= b` if `s` is negative). `i` is a variable of an integer or enumeration type, named by its name alone; `a` and then `b` are evaluated once, before the first round; `s` is a constant other than zero, and 1 if left out. `a` and `b` have `i`'s type, or are untyped constants that take it, or are `INT`s that widen to an `I64` `i`, as with `+` (section 6.1): so in `for i := 0 to LEN(s) - 1`, `i` is an `INT` or an `I64`.

The loop never steps past `b`, so `i` can't wrap: afterwards `i` holds the last value it took, or `a` if the body never ran. The body may assign `i`. Precisely, for a positive `s`: at the end of a round, if `i <= b` and `b - i < s`, the loop ends with `i` as it is; otherwise `i` becomes `i + s`, wrapping, and the loop goes on while `i <= b`. A negative `s` is the same, the other way round.

`break` ends the innermost `while`, `repeat` or `for` it stands in, and the program goes on after it. `continue` ends the round of the innermost one, as reaching the end of its body would: a `while` tests its condition again, a `repeat` tests its `until`, and a `for` steps its variable, or ends. There is no `goto`.

A `continue` passes over the rest of the body, the step of a `while` or a `repeat` included, if it stands there: so the step comes first.

```
while i < LEN(t) do
    ch, n := Str.decode(t[i..LEN(t)]);
    i := i + n;                          (* before any continue *)
    if ch = 0ADH then continue; end;     (* a soft hyphen, passed over *)
    ...
end;
```

7.3 Return

`return e;` ends the procedure with the result `e`; `return a, b;` with several results; `return f(x);` with the several results of `f`, if they are the procedure's own (section 8); `return;` from a procedure without results. A procedure with results that runs off its end returns zeros, each result as a variable of its type starts (section 4).

A slice returned must not point into the procedure's own frame, which is gone once it returns. It is an error to return, as a slice, a local array, a value parameter that is an array, an array literal other than `[]`, or an array a call returns, or a slice of one, a field or an element of one included:

```
proc short(h: Hex): []U8;      (* Hex = [64]U8 *)
begin
    return h[0..8];            (* an error: h is the procedure's copy *)
end;
```

A slice of a `var` parameter, of a module variable, or of what a pointer points at may be returned: none of them is this procedure's frame. A slice of a frame that leaves it some other way isn't caught, and is the exception of section 10: one held in a local slice variable, or in a record, that is returned; one stored in a module variable or through a pointer; and one that came back from a call, as the slice of a `var` parameter does to the caller whose local it is.

7.4 Match

```
match s of
| circle(r):  area := 3.14159 * r * r;
| rect(w, h): area := w * h;
| dot:        area := 0.0;
end;
```

`match` takes a variant, an enumeration, an integer type (a `U8` included) or a string, a `[]U8`; other types are an error. It runs the arm of the case `s` holds. An arm may name the case's fields, in order, all of them; the names stand for those fields of `s` itself, so assigning to one changes `s`; if `s` isn't a variable, they stand for the fields of a copy. An arm may also leave the fields unnamed. Every case must have an arm, and only one, unless there is an `| else:` arm, which must come last.

`match` on an enumeration works the same way: each arm names one of the enumeration's names, or several, separated by commas, for the arm to run when the value is any of them:

```
match op of
| sPlus, sMinus: kind := additive;
| sTimes:        kind := multiplicative;
| else:
end;
```

Every name must have an arm, and only one, unless there is an `| else:` arm. A variant's arm names one case: the fields it may name belong to that case alone.

`match` on an integer runs the arm whose labels hold its value:

```
match ch of
| ' ', '\t':      skip := true;
| '0' .. '9':     digit := true;
| else:
end;
```

A label is a constant, or `lo .. hi`, the constants from `lo` to `hi`, both included (unlike a slice, which leaves `hi` out; `lo` may not be above `hi`). Each must fit the integer's type, and no value may be in two labels. An integer `match` must have an `| else:` arm, which may be empty.

`match` on a string, a `[]U8` or what converts to one (section 5.10), runs the arm one of whose labels it equals, as `=` compares strings (section 6.5):

```
match word of
| "add", "plus": op := add;
| "sub":         op := sub;
| else:          op := none;
end;
```

A label is a string constant, and no two labels may be equal; there are no ranges. A `match` on a string must have an `| else:` arm, which may be empty. Whether an arm names a case or gives labels is set by the type matched: `| Max:` is a case of a variant or an enumeration, and a constant for an integer or a string.

7.5 Order of evaluation

Operands, arguments and the designators of an assignment are evaluated from left to right, and the designators of an assignment before its right side: `a[i()] := g()` calls `i` and then `g`.

7.6 Volatile

A variable reached through a `^VOLATILE T` is volatile, and so is any field or element of it. Each read or write of a volatile variable of a basic, enumeration, set or pointer type is exactly one memory access, of exactly its size, and volatile accesses happen in the order the program writes them. Copying a whole record or array through a volatile pointer is not allowed, since it can’t be one access.

A `^VOLATILE T` doesn’t convert to `^T` without a written conversion, and a volatile variable isn’t passed as a `var` argument, which would lose its volatility.

`FENCE` also orders the program’s ordinary accesses against those of other processors and devices.

Ordinary variables. Only volatile accesses, `CAS`, `XCHG`, `XADD` and `FENCE` are exact. Any other access is ordinary: an implementation may keep a variable in a register, leave out a write that nothing reads, and change the order of ordinary accesses, as long as each thread, on its own, does what this report says. A variable that one thread writes while another reads or writes it, unless both use volatile accesses or the atomic built-ins, or a `FENCE` orders them, holds no particular value then: a read of it gives a value it had or was given, or a mixture of their bytes. Nothing else of the program becomes undefined by it.

7.7 Defer

`defer s;` runs the statement `s` when the program leaves the statement list the `defer` stands in: at its end, or by a `return`, or by a `break` or `continue` out of it. Statements deferred in the same list run in the opposite order to their `defer`s, and only those whose `defer` has been reached run.

```
module Head;
import Sys, Out;

(* Prints the first line of the file at path. *)
proc head(path: []U8): BOOL;
var fd, n, i: INT; buf: [256]U8;
begin
    fd := Sys.openRead(path);
    if fd < 0 then return false; end;
    defer Sys.close(fd);
    n := Sys.read(fd, buf);
    if n < 0 then return false; end;
    for i := 0 to n - 1 do
        if buf[i] = 10 then n := i; break; end;
    end;
    Out.str(buf[0..n]);
    Out.ln;
    return true;
end;

proc main*;
begin
    if not head("/etc/hostname") then Out.str("no hostname\n"); end;
end;

end Head.
```

`return e;` evaluates `e` before the deferred statements run. A deferred statement can't contain `defer`, nor a `return`, `break`, `continue` or `try` that would leave it. A failed check, `Rt.exit` and `Sys.exit` end the program without running the deferred statements still pending.

7.8 Try

A procedure that can fail returns an `ERROR` as its last result (section 5.8), `nil` when it didn't. A caller that can't deal with the failure passes it on to its own caller, which `try` does in a word:

```
proc total(words: [][]U8): (INT, ERROR);
var i, sum: INT;
begin
    for i := 0 to LEN(words) - 1 do
        sum := sum + try number(words[i]);
    end;
    return sum, nil;
end;
```

`try c`, where `c` is a call (below) of a procedure whose last result is an `ERROR`, makes the call and looks at that error. If it is `nil`, the program goes on, and `try c` is what `c` returned without the error. If it isn't, the procedure `try` stands in returns at once, as `return` does (section 7.3): its last result is that error, and every other result is zero, as a variable of its type starts (section 4). Nothing more of the statement `try` stands in is done; in the example, `sum` isn't assigned. So

```
x := try f(a);
```

is, with `e` a variable the program can't name, and `0` standing for a zero of each of the procedure's other results:

```
x, e := f(a);
if e # nil then return 0, e; end;
```

The error is the one `c` returned, and compares with `=` as it did there. `try` has no handler and keeps no state: it is a `return`, written where the call is, so that every place a procedure can leave is in its text. A program that means to deal with an error tests it, as section 5.8 shows.

What `try` is applied to. `c` is a designator that ends in a call: `f(x)`, `Mod.f(x)`, `p^.next(x)`, or a call through a variable of procedure type. Where the procedure has no parameters its parentheses may be left out, as in a statement (section 7.1), and `try` calls it: `try setup;`. A conversion, a built-in procedure (section 6.8) and the case of a variant are written like calls and aren't procedures'; `try` can't be applied to them.

What `try c` is. `c`'s results but the last.

- With none, `try c` isn't a value and stands only as a statement: `try flush;`.
- With one, it is a value of that type, an operand wherever `c` would be: `n := try number(s);`, `sum + try number(s)`, `if try ready(fd) then`. Like `not`, it applies to the factor after it: `try f(x) + 1` is `(try f(x)) + 1`.
- With several, it stands only on the right of a multiple assignment, as `c` itself would, with a target for each: `a, b := try f(x);`.

As a statement, `try c;` drops the values of `c` that it has, and a dropped value is a lint, as a call that drops a result is (section 10); `_ := try c;` and `a, _ := try c;` say it is meant (section 7.1). The error is never dropped: `try` is the way of passing it on.

Where `try` may stand. Every use of `try` is an error unless:

- the procedure it stands in has an `ERROR` as its last result. A module's body, and a procedure with no results, such as `proc main*;`, have none: there an error is dealt with where it is found, by a test;
- the last result of the procedure `c` calls is an `ERROR`;
- it is not inside a deferred statement, which `try` would leave (section 7.7).

A generic module is checked in each instance (section 3), so a procedure of an instance whose last result, by its type arguments, is no `ERROR` and uses `try` is an error in that instance.

Order. The arguments of `c` are evaluated from left to right, then `c` is called, and then its error is looked at (section 7.5). The designators on the left of an assignment are evaluated before its right side as always, so in `a[i()] := try g()`, `i` is called, then `g`; if `g` fails, nothing is assigned. A `try` in the right operand of `and` or `or` is reached only when that operand is evaluated (section 6.2).

Deferred statements. When `try` returns, the statements deferred in the procedure and not yet run are run, the last first, as for a `return` (section 7.7), after the error and the zero results have been chosen; no deferred statement can change them.

8. Procedures

```
ProcDeclaration   = "proc" IdentDef Signature ";"
                    ( "forward" | { LocalDeclaration } "begin" Statements "end" [
ident ]
                    | "code" { [ ident ":" ] [ Instruction ] ";" } "end" [ ident ] ) .
LocalDeclaration  = "const" { ident [ ":" Type ] "=" Expr ";" }
                  | "type" { ident "=" Type ";" }
                  | "var" { ident { "," ident } ":" Type ";" } .
Signature         = "(" [ Params { ";" Params } [ ";" "..." ] ] ")" [ ":" Results ]
                  | [ ":" Results ] .
Params            = [ "var" ] ident { "," ident } ":" Type .
Results           = Type | "(" Type { "," Type } ")" .
```

A procedure without parameters may leave out the parentheses, both where it is declared (`proc main*;`) and where it is called (`Out.ln`). Used without parentheses in an expression, a procedure is a procedure value, not a call.

Parameters. A value parameter is a copy: the procedure may change it, and the caller's variable stays as it was. A `var` parameter is the caller's variable itself; its argument must be a variable of exactly the parameter's type.

Results. A procedure returns nothing, one value, or several. A call that returns several values can only stand on the right of a multiple assignment, alone as a statement, or alone after `return` in a procedure whose results are of the same types, in the same order, which then returns them as they are: `return divmod(n, 2);`. A procedure that can fail has an `ERROR` as its last result (section 5.8), which `try` passes on (section 7.8).

An array a call returns lives to the end of the statement the call is in. A slice of it may be used there, indexed, passed or compared; one kept past the statement, in a variable or anywhere else, points at what is gone, and is the exception of section 10:

```
s := f('a')[0..4];     (* f returns a [8]U8: s is undefined after this *)
Out.str(f('a')[0..4]);   (* used in its statement: defined *)
```

```
module Divide;
import Out;

proc divmod(a, b: I64): (I64, I64);
begin
    return a div b, a mod b;
end;

proc main*;
var q, r: I64;
begin
    q, r := divmod(17, 5);
    Out.int(q); Out.str(" rest "); Out.int(r); Out.ln;
end;

end Divide.
```

forward. `proc f(x: I64): I64; forward;` declares `f` so that it can be called before its body appears; the body comes later in the module, with the same signature, though its parameters may have other names; a

`forward` with no body is an error.

A name after a procedure's `end`, if written, is the procedure's own.

Procedures don't nest.

Code procedures. A procedure whose body is `code` is machine code, written by hand in the target's assembly language (appendix A): one instruction to each `;`, and labels for jumps, up to `end`:

```
proc rdtsc*: U64;
code
    rdtsc;
    shl rdx, 32;
    or rax, rdx;
    ret;
end rdtsc;
```

A label, `name:`, names the place of the instruction after it, within the procedure. A constant of the module may stand wherever a number does. Any bytes may be written with `db`, as in `db 0FH, 31H;`, for an instruction the assembler doesn't know.

It is called like any procedure, with the System V convention: integer and pointer arguments arrive in `rdi`, `rsi`, `rdx`, `rcx`, `r8` and `r9`, `F32` and `F64` ones in `xmm0` to `xmm7`, and the result goes in `rax` (and `rdx` for a second) or `xmm0`. Its parameters and results are therefore of the types C can take (section 9), or slices, at most six integer and eight real parameters, and at most two results. A slice is two integers, as System V passes a struct of them: as a parameter its address arrives in one register and its length in the next, the two counting among the six; as the result its address goes in `rax` and its length in `rdx`, and it is the only result. The instructions are the whole procedure, `ret` included; they must keep `rbx`, `rbp`, `rsp` and `r12` to `r15` as they found them. The compiler doesn't look past what they say.

9. Calling C

```
ExternDeclaration = "extern" ( "proc" IdentDef Signature
                             | "var" IdentDef ":" Type ) .
```

```
link "libX11.so.6";
extern proc XOpenDisplay(name: ^U8): ^U8;
extern proc printf(format: ^U8; ...): I32;
extern var environ: ^^U8;
```

An `extern` procedure or variable is found by its name in the C libraries the program links: the C library always, and each library named by `link`.

An `extern` procedure's parameters and result are integers, `BOOL`, `F32`, `F64`, pointers or procedures; a record goes to C by pointer, never by value. `...` accepts any number of further arguments of those types, and is allowed only in an `extern`. C takes no `float` there, so an `F32` given to `...` is an error: the program writes `F64(x)`. Past the parameters, a real constant goes as an `F64`, an integer constant as an `INT`, and a string literal as a pointer to its bytes followed by a zero byte.

An `extern` variable is C's variable itself: reading it reads C's, assigning it writes C's. It can be of any type, declared as C declares it, a record laid out as C lays it out (section 5.5); the compiler can't check that it is. A C variable that is local to each thread, such as `errno`, can't be reached this way; the C library has a procedure for it (`__errno_location`).

A C union has no raz type. A program reads one as an array of words, and the member it wants through a pointer conversion:

```
var ev: [24]I64;                        (* XEvent: 192 bytes *)
...
XNextEvent(dpy, ADR(ev));
if I32(ev[0]) = KeyPress then
    key := ^XKeyEvent(ADR(ev));
    ...
end;
```

Every raz procedure uses the System V calling convention, so a raz procedure whose parameters and result are of those types can be given to C as a callback.

10. Programs

Any module may export a procedure `main`, declared `proc main*;` or `proc main*: INT;`. The program runs the bodies of all its modules (section 3), then the main module's `main`, if it has one, and ends with status 0, or with the status `main` returns, by calling `Rt.exit`. The `main` of an imported module is an ordinary procedure, which runs only if something calls it; so a library module can carry a `main` that tests it. A program uses C if any of its modules has an `extern` declaration or a `link` of a library; a module's other files, read only to check its interface (section 3), don't count. A program that doesn't is a static executable with no run time besides the code it uses. One that does is linked dynamically and starts through the C library, so that C's own set-up and clean-up run. A program with a module that says `link static;` must not use C, and the compiler reports an error if it does.

Ending. `Rt.exit(status)` ends the program. First it runs the procedures registered with `Rt.atExit`, the last registered first, each at most once; then it ends the program with `status`, through C's `exit` in a program that uses C, so that C flushes its own buffers. A program that stops (below), a failed check (section 6.5) included, runs them too, after its message, so that `Out`'s buffer is written out, and ends with status 2. `Sys.exit` ends the program at once, and a program killed by a signal from outside runs none of them.

```
proc atExit*(p: proc): ERROR;   (* Rt.Full if 32 are registered *)
proc exit*(status: INT);
proc trace*;                    (* the calls under way, on standard error *)
```

`Rt.trace` says on standard error the calls under way, from the one that calls it out, each with its variables, as a program that stops does.

Defined behaviour. Every variable starts at zero, integers wrap, and indexing arrays and slices is checked, so a program's behaviour is defined by this report, with one exception: memory reached other than through a checked variable. Dereferencing a stale or made-up pointer, indexing through a pointer, a slice kept after the call whose frame holds what it points at (section 7.3), a slice of an array a call returned kept past its statement (section 8), an index out of range in a module that says `check off;`, writing a string's bytes, a code procedure

that breaks its rules, and C code can read or write any memory, and what follows isn't defined. Dereferencing `nil`, dividing by zero, calling a `nil` procedure and running out of stack stop the program, on every target, with status 2, as a failed check does. Before it stops, the program prints to standard error where it was, as file, line and column, and why, as far as it can tell: a `nil` pointer, a `nil` procedure, a stack overflow, a bad address or a division fault; then the calls under way, each with its variables of a basic, enumeration, set or pointer type. Whether the machine catches these or the compiled code checks for them is the implementation's choice (appendix A); that they stop the program isn't.

Separate compilation. An importer depends only on a module's interface: what it exports, as an importer sees it, never its source or its code. An implementation that compiles modules on its own makes the same program as if every module were compiled together (appendix A).

Errors. A program that breaks a rule of this report that says it is an error isn't a raz program, and an implementation must reject it: say what is wrong and make no executable. Such a rule is one whose breach would leave the program with no meaning this report can give: a mistake of types, a name that isn't declared or is declared twice, an assignment to what is read-only, a call with the wrong number of arguments, a type name or a procedure with no result used as a value, type arguments given to a module that isn't generic, or too many, an import that can't be found, a procedure declared `forward` and never defined, a module that imports itself. A fault that is only a lint isn't an error: a name declared and never used, or a result that a call standing on its own drops (`Sys.write(fd, s);`, where `_ := Sys.write(fd, s);` says it is meant). A program with one is valid, and an implementation may warn of it. The text of its messages, how many it reports and which warnings it gives are its own (appendix A).

11. Versions

A version of this report is named by the month it was made in and a count: `2026-10.0`, then `2026-10.1` for the next change that month. `raz --version` says which version the compiler follows.

The history of the versions, what each changed, is in `doc/report-history.md`.

Appendix A. What an implementation chooses

Everything in this report binds every implementation of raz, except what follows, which each chooses for itself and documents. This compiler's choices are in `doc/compiler.md`, under "This compiler's choices". The programs in `conform/` are how an implementation shows that it follows this report: it passes them all, its reals aside where its target has none; `conform/check.sh` runs them.

- **The target.** Which machines and systems it compiles for, and the names of their files (`linux-x86_64`, section 3); for each, the size of an address, and so of `INT`, and whether it has reals.
- **Separate compilation.** Whether modules are compiled on their own, and if so into what: the object file, its format, the form of the code in it, when one is out of date, and how an instance is named so that the program keeps one (section 10). This compiler compiles every module of a program together, and writes no object files.
- **Atomics.** How `CAS`, `XCHG` and `XADD` are made sequentially consistent on a machine without atomic instructions (section 6.8): on one with a single processor, by holding off interrupts while they run, for one.

- **The calling convention.** On x86-64 Linux it is System V (sections 8 and 9); elsewhere it is the C library's own, so that `extern` procedures and callbacks work. How a procedure returns several results, and how its frame is laid out, are the implementation's.
- **What is the target's.** Code procedures (section 8) hold the target's machine code, written in its assembly language, and exist only where the implementation says so; `SYSCALL` exists only on Linux targets (section 6.8). A record's layout is the target's C layout (section 5.5), and the order of a value's bytes is the target's (little-endian on x86-64).
- **The assembly language.** How a code procedure's instructions are written (section 8). On x86-64 it is Intel's syntax, as NASM writes it, destination first. Which instructions it knows is the implementation's; `db` gives any others.
- **Sizes in the library.** How many bytes `Out` keeps before it writes them, `Sys.outSize` (the library volume): 4 KB on Linux, fewer on a small machine.
- **The executable.** Its format, and the names and line numbers it carries for a debugger (section 10).
- **How a failure is caught.** A `nil` pointer, a division by zero or a stack overflow may be caught by the machine or by checks in the compiled code; either way the program stops with its message (section 10).
- **Messages and warnings.** The text of the compiler's messages, how many errors it reports before it stops, and which warnings it gives, `-Werror` included (section 10). That a program that is an error in this report is rejected isn't free; the words that say so are. So is what a program that fails says on standard error: conformance is judged by a program's standard output and exit status, never by its standard error.
- **The command line.** How the compiler is run, its options, and where its library lives (sections 1 and 3).

The core. A target may leave out what the system under it can't give: `Sys` past writing, ending the program and getting memory, `Thread`, `link` and `extern`. What it must have, besides the whole language, reals aside, is the core of the library: `Rt`, `Out`, `Mem`, `Arena`, `Lock`, `Str` and `Buf`, and of `Sys` what they need. A program that keeps to the core runs on every target. The rest of the library (`Heap`, `Fmt`, `List`, `Map`, `Vec`, `Sort`, `Math`, `Files`, `Opt`, `Width` and the rest of `Sys`) is the standard library: a target may leave a module out, and one that has it gives what `doc/raz-lib.md` says. `conform/` tests the core only.

Appendix B. The grammar

The whole syntax of raz, gathered from the sections above, with the symbols of section 2. `{ x }` is any number of `x`, `[ x ]` is `x` or nothing, and `( x | y )` is one of them.

```
ident      = letter { letter | digit } .
letter     = "A" .. "Z" | "a" .. "z" | "_" .
digit      = "0" .. "9" .
hexDigit   = digit | "A" .. "F" .
number     = digit { digit }
           | digit { hexDigit } "H"
           | digit { digit } "." digit { digit } [ "E" [ "+" | "-" ] digit { digit } ]
.
char       = "'" ( character | escape ) "'" .
string     = '"' { character | escape | "\x" anyHex anyHex } '"'
           | "`" { rawChar } "`" .
escape     = "\n" | "\t" | "\r" | "\0" | "\\" | "\'" | '\"'
           | "\u{" anyHex { anyHex } "}" .
anyHex     = hexDigit | "a" .. "f" .

Module     = "module" ident [ "(" ModParams ")" ] ";"
             { Import } { Link } [ Check ] { Declaration }
             [ "begin" Statements ] "end" ident "." .
ModParams  = ident { "," ident } { ";" ident ":" "proc" Signature } .
Import     = "import" ImportItem { "," ImportItem } ";" .
ImportItem = ident [ ":=" ident [ "(" Type { "," Type } ")" ] ] .
Link       = "link" ( string | "static" ) ";" .
Check      = "check" "off" ";" .

Declaration = "const" { IdentDef [ ":" Type ] "=" Expr ";" }
            | "type" { IdentDef "=" Type ";" }
            | "var" { IdentList ":" Type ";" }
            | ProcDeclaration ";"
            | ExternDeclaration ";" .
IdentDef    = ident [ "*" ] .
IdentList   = IdentDef { "," IdentDef } .

Type      = QualIdent
          | "[" Expr "]" Type
          | "[" "]" Type
          | "^" [ "VOLATILE" ] Type
          | [ "packed" ] "record" { IdentList ":" Type ";" } "end"
          | "variant" { Case ";" } "end"
          | "(" ident { "," ident } ")"
          | "set" "of" QualIdent
          | "proc" Signature .
Case      = ident [ "(" FieldList { ";" FieldList } ")" ] .
FieldList = ident { "," ident } ":" Type .
QualIdent = [ ident "." ] ident .

Expr       = Conjunct { "or" Conjunct } .
Conjunct   = Relation { "and" Relation } .
Relation   = SimpleExpr [ RelOp SimpleExpr ] .
RelOp      = "=" | "#" | "<" | "<=" | ">" | ">=" | "in" .
SimpleExpr = [ "-" ] Term { AddOp Term } .
AddOp      = "+" | "-" | "|" | "xor" .
Term       = Factor { MulOp Factor } .
MulOp      = "*" | "/" | "div" | "mod" | "&" | "shl" | "shr" .
Factor     = number | char | string | Designator | "(" Expr ")"
           | "not" Factor | "~" Factor
```

```
           | "[" [ Expr { "," Expr } ] "]"
           | "{" [ Element { "," Element } ] "}"
           | "^" [ "VOLATILE" ] Type "(" Expr ")"
           | "try" Designator .
Element    = Expr | ident ":" Expr .
Designator = QualIdent { Selector } .
Selector   = "." ident | "^" | "[" Expr [ ".." Expr ] "]"
           | "(" [ Expr { "," Expr } ] ")" | "." "(" Type ")" .

Statements = { Statement ";" } .
Statement  = Designator [ ":=" Expr ]
           | "_" ":=" Expr
           | Target "," Target { "," Target } ":=" Expr
           | "if" Expr "then" Statements
             { "elsif" Expr "then" Statements } [ "else" Statements ] "end"
           | "while" Expr "do" Statements "end"
           | "repeat" Statements "until" Expr
           | "for" ident ":=" Expr "to" Expr [ "by" Expr ] "do" Statements "end"
           | "match" Expr "of" { "|" Arm } "end"
           | "return" [ Expr { "," Expr } ]
           | "break"
           | "continue"
           | "defer" Statement
           | "try" Designator .
Arm        = ( ident [ "(" ident { "," ident } ")" ]
             | Label { "," Label } | "else" ) ":" Statements .
Label      = Expr [ ".." Expr ] .
Target     = Designator | "_" .

ProcDeclaration   = "proc" IdentDef Signature ";"
                    ( "forward" | { LocalDeclaration } "begin" Statements "end" [
ident ]
                    | "code" { [ ident ":" ] [ Instruction ] ";" } "end" [ ident ] ) .
LocalDeclaration  = "const" { ident [ ":" Type ] "=" Expr ";" }
                  | "type" { ident "=" Type ";" }
                  | "var" { ident { "," ident } ":" Type ";" } .
Signature         = "(" [ Params { ";" Params } [ ";" "..." ] ] ")" [ ":" Results ]
                  | [ ":" Results ] .
Params            = [ "var" ] ident { "," ident } ":" Type .
Results           = Type | "(" Type { "," Type } ")" .
ExternDeclaration = "extern" ( "proc" IdentDef Signature
                             | "var" IdentDef ":" Type ) .
```

`character` is any character but a line break, and in a string any but `"` and `\`, in a character any but `'` and `\`. `rawChar` is any character but `` ` ``, line breaks included. A procedure argument of a generic module (section 3) is written as a `QualIdent`, which is a `Type` here. `...` comes last in a `Signature`, and only in an `extern`. An `Instruction` is written in the target's assembly language (appendix A).