Perl 6 Rigorous Technical Specification

A Comedy of Errors

Author: John M. Długosz
dhcgnd702@sneakemail.com

contents, etc.

Table of Contents

1 General....11
1.1 Notation....11
1.1.1 Typographical conventions....11
1.1.2 Normative or Informative....11
1.2 Observable Behavior....11
1.3 ®noncen....11
1.4 Grammar....12
2 Definitions....13
3 Lexical Conventions....15
3.1 Character Set....15
3.1.1 Normalization....15
3.1.2 Byte-Ordering Mark....15
3.1.3 Encoding Directive....16
3.1.4 Conflict with #! mark....16
3.1.5 encoding pragma....17
3.1.6 Out-of-band encoding specification....17
3.2 Whitespace....17
3.3 Unspace....18
3.4 Line Counting....20
3.5 Bracketing Characters....20
3.6 Comments....21
3.6.1 POD Sections....21
3.6.2 The “#” character....21
3.6.2.1 Single-line comments....21
3.6.2.2 Embedded comments....22
3.7 Tokens....23
3.8 Identifiers....23
3.8.1 Identifiers for internal use....23
3.8.2 Semi-reserved names....23
3.8.3 Other future names....24
3.9 Keywords....24
3.9.1 Table of keywords....24
3.9.2 Reserved trait names....27
3.10 Operators and Punctuators....28
3.11 Literals....28
3.11.1 Integer....28
3.11.2 Floating Point....28
3.11.3 Infinities....28
3.11.4 Quoted Strings....28
4 Basic Concepts....30
4.1 Translation and Execution....30
4.2 Episodes....30

4.2.1 Exception handling notes....30
4.2.2 Context of Execution....30
4.2.3 Introspection....31
4.2.4 Parsing episode....31
4.2.5 blah blah....31
4.2.6 BEGIN episode....31
4.2.7 CHECK episode....31
4.2.8 INIT episode....32
4.2.9 START episode....32
4.2.10 FIRST episode....32
4.2.11 PRE episode....32
4.2.12 ENTER episode....33
4.2.13 normal execution episode....33
4.2.14 NEXT episode....33
4.2.15 CATCH episode....33
4.2.16 CONTROL episode....34
4.2.17 LEAVE episode....34
4.2.18 KEEP episode....34
4.2.19 UNDO episode....34
4.2.20 POST episode....34
4.2.21 LAST episode....35
4.2.22 END episode....35
4.3 Abstract Syntax Tree....35
4.4 Start and Termination....35
4.5 Memory Model....35
4.6 Scope....35
4.6.1 Package-qualified names....35
4.6.1.1 Compile-time resolution....36
4.6.1.2 Run-time resolution....36
4.6.2 Pseudo-packages....36
4.7 Name Lookup....37
4.8 Sigils ($, @, %, etc.)....37
4.9 Lvalues....39
4.10 Closures....39
5 Other Conceptual Notes....40
5.1 One-pass parsing....40
5.2 dots....40
5.3 Undefined types....41
5.4 Even more miscellaneous....41
6 Execution Context....42
7 Terms....43
7.1 Variables and Sigils....43
7.1.1 Code Objects....43
7.2 * “Whatever”....43
7.3 Captures....44
7.4 Signatures....45

8 Expressions....47
8.1 Postfix and Postcurmfix....47
8.2 Subscripting....47
8.2.1 Types of Braces....47
8.2.2 Item or Slice return....47
8.2.3 Multidimensional Subscripts....47
9 Statements....48
10 Declarations....49
10.1 Binding....49
10.2 Hierarchical types....49
10.3 Hashes....49
10.4 Pseudo-assignment....49
10.5 Variable Initialization....50
10.5.1 * (“whatever”) initialization....50
10.5.2 Default Initialization....50
1.1.1.1 Item Variables....50
1.1.1.2 Array and Hash Variables....51
10.6 Subroutines....51
10.6.1 Kinds of Subroutines....51
10.6.1.1 sub....51
10.6.1.2 method....52
10.6.1.3 submethod....52
10.6.1.4 operators....52
10.6.1.5 conversions....52
10.6.2 Parameters....53
10.6.2.1 Generic types....53
10.6.3 Return type....53
11 Closures....56
11.1 in-place blocks....56
11.2 Closures vs Routines....56
11.3 Literal Hashes....57
11.4 Cloning....57
11.5 “closing” over the symbols....59
11.6 Episodic blocks....60
11.6.1 Episodic blocks....61
11.6.2 63
11.6.3 Semantics....63
11.6.4 Expressions....63
11.6.5 Setting on variables....63
11.6.6 Explicit use of traits....64
11.6.7 Use in top-level code....64
12 The Type System....65
12.1 Junction Types....65
12.2 Typing with Variables, Containers, and Values....66
12.3 Static Type Checking....66
12.3.1 Conceptual Overview....66

12.3.1.1 Effects of Typing....66
12.3.1.2 Errors in type....67
12.3.1.3 Type Conversions....67
12.3.1.4 Static and Dynamic type....68
12.3.1.5 Generic Types....68
12.3.1.6 Unspecified Types....69
12.4 Fully-Specified Types....69
12.5 Generic Types....70
12.5.1 Class names used in methods....71
12.5.2 Error Checking and specialization....74
12.5.3 Code object identity and specialization....75
12.6 The Failure type....76
12.7 Type equivalence....77
12.8 Type checking....77
12.9 Container type....78
12.10 Variable Binding....79
12.11 Parameter Binding....80
13 Signatures and Captures....81
13.1 What the Synopses Says....81
13.2 Variable Declarations....83
14 Packages and Modules....84
15 Classes....85
15.1 Items....85
15.1.1 𝒥-items....85
15.1.2 □-items....85
15.1.3 𝒜-items....85
15.1.3.1 Declarators....86
15.1.3.2 class traits....86
15.2 Special Members....86
16 Roles....87
16.1 The CUB paper....87
16.2 Parsing and storing a Role....88
16.3 Compositing....89
16.3.1 Semantics of compositing....89
16.3.1.1 Removing duplicate Roles....90
16.3.1.2 Building the composite....90
16.3.1.3 Reference to not-copied items....91
16.3.1.4 Realization of the Composite....91
16.3.2 Resolving conflicts....92
16.3.2.1 Priority....92
16.3.2.2 Exclusion and Adjustment....93
16.3.2.3 Redefining....93
17 Calls....94
17.1 sub calling lookup....94
17.2 ordinary method calling lookup....94
17.3 operator notation calls....95

17.4 private method calling lookup....96
17.5 indirect method call....96
17.6 method candidate selection....96
17.7 method call semantics....96
17.8 sub candidate selection....96
18 Grammars....97
19 Exceptions....98
19.1 Standard Exception Types....98
20 Macros....99
21 POD documentation....100
22 Garbage Collection....101
23 Special Variables....102
23.1 Magical lexically-scoped values....102
23.1.1 &?BLOCK....102
23.1.2 $?LINE....102
23.1.3 &?ROUTINE....102
24 Library....103
24.1 ubiquitous functions....103
24.1.1 caller....103
24.1.2 callsame....103
24.1.2.1 How might it work....104
24.1.2.2 Also works in method candidates....105
24.1.3 callwith....106
24.1.4 context....106
24.1.5 leave....106
24.1.6 nextsame....106
24.1.7 nextwith....106
24.1.8 return....106
24.1.9 self....107
24.1.10 temp....107
24.1.11 want....108
24.1.12 VAR....108
24.2 Any (class)....108
24.3 Array (class)....108
24.3.1 any....108
24.3.2 bytes....108
24.3.3 chars....108
24.3.4 codes....109
24.3.5 elems....109
24.3.6 graphs....109
24.3.7 push....109
24.4 AST....109
24.5 Bag....109
24.6 bit (compact type)....109
24.7 Bit (class)....109
24.8 Blob (class)....109

24.9 Block (class)....109
24.10 bool (compact type)....110
24.11 Bool (enum)....110
24.12 Buf and buf....110
24.13 Capture (class)....111
24.14 Class (class)....111
24.15 Comparable (role)....111
24.15.1 after, !after, before, !before....111
24.15.2 complete....112
24.15.3 cmp....113
24.16 ComparableNumeric....113
24.16.1 < , > , <= , >=....113
24.16.2 complete....114
24.16.3 <=>....114
24.17 Context (role)....115
24.17.1 context....115
24.17.2 caller....115
24.17.3 file....115
24.17.4 hints....115
24.17.5 inline....115
24.17.6 leave....115
24.17.7 line....116
24.17.8 my....116
24.17.9 package....116
24.17.10 want....116
24.18 Code (role)....116
24.18.1 arity....116
24.18.2 assuming....116
24.18.3 callwith....117
24.18.4 labels....117
24.18.5 signature....117
24.19 Complex (class)....117
24.20 Encoding (class)....118
24.21 Exception (class)....118
24.22 Failure (role)....118
24.23 GLOBAL (package)....118
24.24 Grammar (class)....118
24.25 Hash (class)....118
24.26 Infinite....118
24.26.1 Roles....119
24.26.2 −....119
24.26.3 <=>, cmp....120
24.26.4 cardinal....120
24.26.5 negative....120
24.26.6 new....120
24.26.7 positive....121

24.27 int (compact type)....121
24.28 Integral (role)....121
24.29 Int (class)....121
24.29.1 operator div....121
24.29.2 abs....122
24.30 IO (role)....122
24.30.1 close....122
24.31 Junction (class)....122
24.32 KeyBag (class)....122
24.33 KeyHash (class)....123
24.34 KeySet (class)....123
24.35 KitchenSink (role)....123
24.35.1 clear....124
24.35.2 push....124
24.36 List (class)....124
24.37 Macro (class)....124
24.38 Mapping (class)....124
24.39 Match (class)....124
24.40 Metaobject (role) aka Type? 124
24.40.1 convert:<Str>....124
24.40.2 attributes....125
24.40.3 bless....125
24.40.4 buts....125
24.40.5 can....126
24.40.6 does....126
24.40.7 isa....126
24.40.8 methods....126
24.40.9 name....126
24.40.10 names....127
24.41 Method (class)....127
24.42 Module....127
24.43 MY (pseudo-package)....127
24.44 Num (class)....127
24.45 Object (class)....128
24.45.1 BUILD....128
24.45.2 BUILDALL....128
24.45.3 CANDO....128
24.45.4 clone....128
24.45.5 DESTROY....128
24.45.6 DESTROYALL....128
24.45.7 fmt....128
24.45.8 HOW....128
24.45.9 new....129
24.45.10 perl....129
24.45.11 WALK....129
24.45.12 WHAT....130

24.45.13 WHEN....130
24.45.14 WHENCE....130
24.45.15 WHICH....130
24.45.16 WHO....130
24.46 Ordinal (role)....130
24.46.1 ++....130
24.46.2 predecessor....131
24.46.3 strict....131
24.46.4 successor....132
24.47 Order (enum)....132
24.48 P6opaque (class)....132
24.49 Package (class)....132
24.50 Pair (class)....132
24.51 Range (class)....132
24.52 Rat (class)....132
24.53 Regex (class)....133
24.54 Role (class)....133
24.55 Routine (class)....133
24.55.1 as....133
24.55.2 do....133
24.55.3 multi....133
24.55.4 name....133
24.55.5 of....134
24.55.6 signature....134
24.55.7 unwrap....134
24.55.8 wrap....134
24.55.8.1 usage notes....136
24.56 Scalar (class)....136
24.57 Seq (class)....136
24.58 Set (class)....137
24.59 Signature....137
24.59.1 arity....137
24.59.2 count....137
24.59.3 item....137
24.59.4 list....137
24.59.5 rw....138
24.59.6 void....138
24.60 Str (class)....138
24.60.1 Str::Canonicalization (type)....139
24.60.2 bytes....139
24.60.3 chars....140
24.60.4 codes....140
24.60.5 graphs....140
24.61 StrPos (class)....140
24.62 Sub (class)....140
24.63 Submethod (class)....140

24.64 SUPER (pseudo-package)....141
24.65 Tie (module)....141
24.65.1 Tie::Array (role)....141
24.65.2 Tie::Hash (role)....141
24.65.3 Tie::Scalar (role)....141
24.66 Whatever (class)....141
24.66.1 multidimensional....142

1 General

1.1 Notation

1.1.1 Typographical conventions

Because the ellipses is a legal Perl 6 token, omitted text or an indication that some other lines are present is indicated with the

°°°

mark. That way it will not be confused with class C { ... } which is meant to actually be ellipses in the listing.

1.1.2 Normative or Informative

A tag of “Informative”, typeset at the right margin like this:

Informative

means that what follows is not part of the normative reference, but merely informative. It may offer insight or duplicate information defined elsewhere.

Such labels go out of scope at the next numbered heading.

Footnotes are non-normative.

1.2 Observable Behavior

The “meaning” of a Perl 6 program is given by its observable behavior. All conforming implementations will, given the same program and subject to the same input, produce the same observable behavior.

Observable behavior is defined in terms of calls to external functions with side effects, and reading and writing to low-level memory as described by low-level types and buffers.

> Need to define a low-level access to raw memory. Suggest a function on raw types and buffers that returns an integer (a pointer address), and once done so, becomes pinned and ‘volatile’ to some extent.

1.3 ®nonce$_n$

Often, the behavior of some language feature or syntax is described as being “equivalent to” some other code that does not make use of the feature in question or is otherwise simpler.

But such equivalence would also be qualified with “except that $whatever is an anonymous generated variable that is not present in any scope.”

Instead, the use of a symbols named *®nonce$_n$* implies this. Identifiers beginning with

"®" are meant for internal implementation use only, so this really could be how the implementation generates code for the construct, and it tells you that this is not a symbol you should pay attention to, if you even perceive it at all. The name "nonce", set in italic, means that a unique name is actually generated. If necessary, a number will follow, so one example can have `nonce1`, `nonce2`, etc. Those are to be taken as different unique names.

1.4 Grammar

The grammar fragments shown for the various syntactic constructs is meant to be correct and comprehensive, but is not written in a fully formal notation. The official grammar for Perl 6 is given in the Appendix.

> During the current development stage, this is STD.pm which lives separately from this document's draft.

Any conflicts between the informal grammar fragments and the official grammar that can't be attributed to the informality is an error that needs to be addressed.

2 Definitions

attribute

circumfix

closure block — A list of statements inside curly braces is formally known as a closure block, to distinguish it from the informal use of block as in "a block of code" or "a comment block". The syntactic construct or the compiled object representing it may be abbreviated to just "closure" or "block" if the context is clear.

consistent types — See page 79. In type checking, type A is consistent with type B if objects of type B can always be used where type A is expected. This is true if B extends A polymorphically so it holds an "isa" relationship. Junctions complicate things.

episode

elaborate — need to define what "elaborating a variable" means exactly, or replace uses of the term with the eventual formal nomenclature.

function — The word "function" is used informally. Perl 6 functions are any callable code whether it returns a value or not.

hash

immutable type — Objects with these types behave like values, i.e. `$x === $y` is true if and only if their types and contents are identical (that is, if `$x.WHICH eqv $y.WHICH`). Types are made to be immutable types by defining a `WHICH` method that returns a value that is not unique to the instance but is the same if the value is the same.

mutable type — Objects with these types have distinct `.WHICH` values that do not change even if the object's contents change.

overlapping types — See page 80.

postcircumscript

propertes - Properties are like object attributes, except that they're managed by the individual object rather than by the object's class.

routine declarator — The keywords `method`, `macro`, `regex`, `rule`, `sub`, `submethod`, `token` used to introduce a declaration.

scoping declarator — The keywords `constant`, `has`, `my`, `our`, `state`, used to introduce a declaration. The keywords `temp` and `let` are *not* declarators.

sigil

trait — Properties applied to objects constructed at compile-time, such as variables and classes, are also called **traits**. Traits cannot be changed at run-time. Changes to run-time properties are done via mixin instead, so that the compiler can optimize based on declared traits.

translation unit — A source file fed to an implementation as the top-level work item is a translation unit, as is a string fed to an embedded implementation. A file processed with the require or use statements is a translation unit. A string given to eval is a translation unit.

Closures within the above translation units, or Perl code embedded inside literal strings or regexes, are not separate translation units.

3 Lexical Conventions

3.1 Character Set

Perl 6 source text is defined in terms of Unicode characters. Regardless of the source text's physical encoding (whether some Unicode encoding format such as UTF-8, or another character encoding system entirely), it is transcoded to a stream of Unicode code points as it is streamed in, and the parser operates on the resulting stream.

An embedded Perl implementation might be called with text to translate and/or execute. Encoding issues must be defined as part of that programmatic interface.

3.1.1 Normalization

Unicode normalization form NFKC is used outside of string literals, comments, and POD.

In particular, identifiers, keywords, and other tokens are matched even if they are encoded differently with respect to issues included in NFKC normalization. For example,

```
constant ṩ = 3.872;
```

is written in a module, where the identifier ṩ matches the domain use of that notation, so everyone likes it. However, when people use the module, they type the letter ṩ using various different systems and text editors configured in different ways. It just so happens that there are many ways this can be encoded in the source file that all have identical meaning in Unicode. It would be dreadful if all the files *looked* the same but did not match according to the Perl 6 implementation.

However, inside string literals, the encoding is not changed from what was read from the file or produced by the encoding module if the file is not stored in a Unicode encoding form. This is important so that deliberate choices of ligatures or composition are not destroyed.

> Note that the standard functions for comparing strings needs to deal with this, too. This should be explained at length in the proper section. Two strings can be unequal in containing a different sequence of code points, while still being equal in containing the same sequence of graphemes.

3.1.2 Byte-Ordering Mark

A file may begin with a U+FEFF character, encoded in the manner of the rest of the file. The implementation will recognise the bytes produced by encoding U+FEFF in any of UTF-8, UTF-16 LE, UTF-16 BE, UTF-32 LE, or UTF-32 BE and infer the file encoding from this.

3.1.3 Encoding Directive

When a Perl 6 implementation reads program text from a file, the file is expected to be UTF-8 or a proper subset[1] by default. The file may begin with an encoding directive like

```
=encoding Windows-1252
```

to indicate the encoding of that file. If a byte-ordering mark (BOM) is present, the '=' character must immediately follow and the specified encoding be consistent with that inferred from the byte-order mark.

Otherwise, the '=' must be the first character in the file, with no whitespace before it. The string "=encoding" is itself encoded in the specified encoding system, so for encodings that do not have ASCII as a subset special matches are needed to recognise the string. In such cases, the specified encoding must be consistent with what was inferred from the encoding of that string.

After the string "=encoding" are 1 or more characters that may be ordinary spaces or tabs (equivalent to U+0020 and U+0009), followed by the name of the encoding, followed optionally by more spaces and tabs. The line ending is encoded in the proper manner for that character encoding, and if necessary, may be taken as a canonical example of how lines will be separated in the rest of the file.

The name of the encoding (also itself encoded in that encoding) is any encoding specification that is allowed by the Encoding class (see page 120).

The rest of the file is read using this encoding.

An implementation may support additional encoding directives in the file to indicate a change of encoding. So, a portable program may not contain additional lines of that form as it may be meaningful on an enhanced implementation and be ignored on others. Because of that, an implementation may issue a warning message if subsequent encoding directives are encountered.

3.1.4 Conflict with #! mark

Informative

Both the BOM and the =encoding directive may conflict with the "magic number" feature used by some operating system shells, where the file must begin with the two bytes 0x23 0x21. The "magic number" only works if the first line of the file is in ASCII anyway, and is interpreted as "#!" followed by the file name of the Perl 6 implementation.

So if you can use that feature, you can use the "use encoding" pragma too.

If the file is in EBCDIC or UTF-16, you would have to use the "=encoding" directive

1 In particular, Latin-1 (a.k.a. ISO 8859-1), and ASCII are proper subsets of UTF-8. A program reading UTF-8 will have no trouble if fed either of those. Note that Windows-1252 ("Western") and other Windows code pages are *not* proper subsets of UTF-8.

for EBCDIC or the BOM and/or the "=encoding" directive for UTF-16.

3.1.5 encoding pragma

The "=encoding" directive is a different feature from the "use encoding" pragma, and independent from it. There are some important differences in how they operate.

The "=encoding" directive itself appears in the encoding of the file that it specifies. This allows it to work with any encoding that the implementation is aware of, even if it doesn't have ASCII as a subset.

The "use encoding" pragma and its arguments will be read, like any other statement, in the current encoding. The compile-time execution it specifies will change the encoding, which takes effect with the bytes immediately following. To use it to specify the encoding of the file as a whole, the statement itself must be readable in the default encoding (and the following character interpreted as something suitable, if the parser reads ahead), which means that it must have ASCII as a subset.

The "use encoding" pragma is block scoped. This allows it to be used to paste in source text without transcoding it to match the rest of the file.

The argument to "use encoding" is a string specifying the name of the encoding, and may be any encoding specification that is allowed by the Encoding class (see page 120).

> We have yet to specify exactly when the parser switches to the new encoding. After the arguments are read, and including a terminating symbol such as a semicolon? This should be mused over by those having an eye to implementing it, and specified in terms of the standard grammar productions. It should be possible to switch from, say, ASCII to EBCDIC and know exactly which character is read in which encoding. An ASCII space would be interpreted as an EBCDIC control code, and an EBCDIC space would be interpreted as an ASCII '@'.

3.1.6 Out-of-band encoding specification

If the implementation is given the name of a text file to execute as Perl 6 source code, there may be an implementation-dependent way to specify the encoding along with the file name.

All standard Perl 6 functions that read source code from a file will take an optional :encoding named argument or adverb, in the same manner as the open function.

3.2 Whitespace

Any characters having the Unicode property of "White Space" counts as whitespace. This includes typographical entities such as thin spaces.[2]

2 So if you use a U+200A HAIR SPACE between "@x" and "[2]", the Perl compiler will not be confused, but any human reader will not see the significant use of whitespace.

Comments count as whitespace.

Unlike most contemporary languages, whitespace is *sometimes* significant.

informative

- Whitespace disambiguates infix and postfix operators: postfix operators may never have intervening space, and if a token can mean either a postfix or infix operator, if there is no intervening space it is taken as the postfix and if there is whitespace it is taken as the infix.
- Whitespace is not allowed before postcircumscript operators, including member access "dot", subscripting, and function call.
- Whitespace is not allowed between components of a name, including around the scope operator "::" and after the sigil.

3.3 Unspace

Any contiguous whitespace (including comments) may be hidden from the parser by prefixing it with "\" (U+005C Reverse Solidus , a.k.a. Backslash). This construct is known as the "unspace".

An unspace provides for visual space or comments where Perl would not allow whitespace. Although not considered whitespace in those situations where whitespace would be significant, it is still a token separator and thus can't be used in the middle of a token.

A "\" where a postfix is expected is also an unspace. That is,

```
$subref\($arg)
```

treats the lone "\" as an unspace, as opposed to escaping out the following character as "\("

A postfix is never expected after a space, so

```
foo \($arg)
```

is parsed as a list operator with a Capture argument. That is, the "\(" has its normal meaning. Conversely, a unary "\" cannot be followed by whitespace, or it would be taken as an unspace instead.

Comments are recognized inside an unspace, and itself becomes whitespace that is hidden by the unspace.

```
$object\   #comment
       .say
```

An unspace may contain a comment, but a comment may not contain an unspace. In particular, end-of-line comments do not treat backslash as significant. The construct:

```
#\ (◦◦◦
```

it is an end-of-line comment, not an embedded comment. Write:

```
\ #(
      ∘∘∘
    )
```

to mean an embedded comment inside an unspace.

informative

Since Perl requires an absence of whitespace between a noun and a postfix operator, using unspace lets you line up postfix operators:

```
%hash\   {$key}
@array\  [$ix]
$subref\ ($arg)
```

Other postfix operators may also make use of unspace:

```
$number\  ++;
$number\  --;
1+3\      i;
 $object\  .say();
$object\#{ your ad here }.say
```

Another normal use of a you-don't-see-this-space is typically to put a dotted postfix on the next line:

```
$object\  # comment
.say

$object\#[ comment
].say

$object\
.say
```

Notice that comments are recognized inside an unspace, and itself becomes whitespace that is hidden by the unspace.

But unspace is mainly about language extensibility: it lets you continue the line in any situation where a newline might confuse the parser, regardless of your currently installed parser. (Unless, of course, you override the unspace rule itself.)

An unspace can be used to break a long line that contains a heredoc, where the heredoc text is expected on the “next” line:

```
ok(q:to'CODE', q:to'OUTPUT', \
"Here is a long description", \ # --more--
todo(:parrøt<0.42>, :dötnet<1.2>));
   ∘∘∘
```

```
CODE
◦◦◦
OUTPUT
```

To the heredoc parser that just looks like:

```
ok(q:to'CODE', q:to'OUTPUT', "Here is a long description", todo(:parrøt<0.42>, :dötnet<1.2>));
◦◦◦
CODE
◦◦◦
OUTPUT
```

Note that this is one of those cases in which it is fine to have whitespace before the unspace, since we're only trying to suppress the newline transition, not all whitespace as in the case of postfix parsing.

3.4 Line Counting

The implementation notes the current line number of a file that is being parsed, available as $?LINE (see *Magical lexically-scoped values* on page 104).

A file starts on line 1, and each occurrence of a line-break sequence (longer forms take priority over shorter) increments the line number by 1, and is recognized as the end-of-line for purposes where the end of the line is significant.

Lines are counted even if the end-of-line sequence is inside an unspace.

The list of end-of-line sequences are: of U+000D U+000A, U+000D, U+000A, U+2028, U+2029.

By default, U+2028 LINE SEPARATOR and U+2029 PARAGRAPH SEPARATOR also each count as a line separator. However, the implementation shall provide a way to customize this, adding characters to the list of line-break sequences, enabling a specified sequence of characters to count as one line break, or removing items from this list. This allows the Perl parser to agree with other tools on line numbers.

All characters in a line-break sequence must have the Unicode property of “White Space”. [3]

3.5 Bracketing Characters

Perl 6 uses “bracketing characters” for some purposes, and allows you to use any pair of bracketing characters that don’t otherwise have a special meaning.

Bracketing characters are defined as any Unicode characters with either bidirectional mirrorings or Ps/Pe properties.

3 This does not preclude an implementation from updating its Unicode tables with properties for a user-defined character or otherwise customizing the Unicode database, in order to allow some oddball end-of-line code. The point is, the grammar engine will agree that those characters are indeed whitespace.

Characters with no corresponding closing character do not qualify as opening brackets. This includes the second section of the Unicode BidiMirroring data table, as well as U+201A and U+201E.

If a character has Ps or Pe properties, then then any entry in BidiMirroring is ignored (both forward and backward mappings).

For any given Ps character, the next Pe codepoint (in numerical order) is assumed to be its matching character even if that is not what you might guess using left-right symmetry. Therefore U+298D maps to U+298E, not U+2990, and U+298F maps to U+2990, not U+298E. Neither U+298E nor U+2990 are valid bracket openers, despite having reverse mappings in the BidiMirroring table.

The U+301D codepoint has two closing alternatives, U+301E and U+301F; Perl 6 only recognizes the one with lower code point number, U+301E, as the closing brace. This policy also applies to new one-to-many mappings introduced in the future.

A program can easily learn Perl's interpretation of whether a given character is an opening or closing bracketing character and if so what it's mate is, using the ?????

Need to define this in standard patterns and related library functions.

3.6 Comments

3.6.1 POD Sections

A POD section (see *POD documentation* on page 102) may be used as a multi-line comment. Any unrecognized POD format will serve as a comment block. As far as the Perl 6 parser is concerned, any POD section is a comment, but some POD blocks have meanings to other tools.[4]

The POD =for construct defines a POD section on one line, so it is also a one-line comment.

POD is recognized on any line, regardless of the state of the parser. This means that POD will "nest" inside multi-line comments, string literals, etc.

3.6.2 The "#" character

Except within a string literal or POD section, a "#" character always introduces a comment.

There are two forms of comments that may be introduced with the "#" character: embedded and single-line

3.6.2.1 Single-line comments

If the "#" character is not immediately followed by an opening-bracket character, it

4 That is, you would not want source comments to appear mixed in with generated class documentation, for example.

comments out everything from the # to the end of the line, not including the end-of-line itself.

Note that '\' immediately following the '#' is not treated as an unspace construct, because unspace is not recognized inside comments. See also §3.3 on page 18.

3.6.2.2 Embedded comments

Embedded comments are supported as a variant on quoting syntax, introduced by # plus any user-selected bracket characters (as defined in *Bracketing Characters* on page 20):

```
say #( embedded comment ) "hello, world!";
$object\#{ embedded comments }.say;
$object\ #「
        embedded comments
」.say;
```

Brackets may be nested, following the same policy as ordinary quote brackets (see *Quoted Strings* on page 28).

There must be no space between the # and the opening bracket character.[5]

An embedded comment is not allowed as the first thing on the line.

```
#sub foo       # line-end comment
#{             # ILLEGAL, syntax error
#   ◦◦◦
#}
```

If you wish to have a comment there, you must disambiguate it to either an embedded comment or a single-line comment. You can put a space in front of it to make it an embedded comment:

```
#sub foo              # line end comment
  #{                  # okay, comment
  ◦◦◦                 # extends
}                     # to here
```

Informative

Or you can put something other than a single # to make it a single-line comment. Therefore, if you are commenting out a block of code using the single-line comment form, we recommend that you use ##, or # followed by some whitespace, preferably a tab to keep any tab formatting consistent:

```
##sub foo
##{                  # okay
##   ◦◦◦
##}
```

5There may be the *visual appearance* of space for some double-wide characters, however, such as the corner quotes above.

```
# sub foo
# {                 # okay
#   ◦◦◦
# }


#     sub foo
#     {             # okay
#       ◦◦◦
#     }
```

However, it's often better to use pod comments because they are implicitly line-oriented. And if you have an intelligent syntax highlighter that will mark pod comments in a different color, there's less visual need for a # on every line.

3.7 Tokens

3.8 Identifiers

Identifiers are Unicode character sequences as defined Default Identifier Syntax as described by Unicode Standard Annex #31 (http://www.unicode.org/reports/tr31/).

Modified where things don't have to begin with a letter.

3.8.1 Identifiers for internal use

In addition, identifiers may begin with the non-letter character "®", U+00AE Registered Sign, for use in implementing the language-support library or other use by the implementation.

A program may not use the ® character normally as part of an identifier, as it is not a letter. But it may appear in generated code, and generated code and library symbols use it so as not to conflict with the program. The implementation may use a pragma to enable the use of this special character in a lexical scope.

So, a program enumerating a scope may see, along with the expected items of `$x`, and `@y`, things like `$®366429` and `&®current_handler`. The various functions that provide introspection might or might not filter these out, or filter them sometimes, incompletely, and inconsistently. The program should ignore them if they are revealed.

3.8.2 Semi-reserved names

Names composed entirely of all capital unaccented ASCII letters may be introduced in future versions of Perl 6, without providing explicit backward-compatibility

features for existing code.

Be careful if you use such an “all-cap name” in your program.

Names of pseudo-packages are reserved (at minimum) in the first position of a package-qualified name or type name using the :: sigil. Other all-cap names may be used as pseudo-packages in the future that will be reserved (only) in the first position of a package-qualified name or type name using the :: sigil. So, never use an all-cap name as a top-level package or type name with a single :: sigil and no other qualification.

All-cap names may also be used in the future as episodic blocks like BEGIN. So never use such a word at the beginning of a statement and follow it with a block.

All-cap names may be used in the future as member functions with semantics dictated by the implementation, such as BUILD. So never use an all-cap name for a method or submethod name.

All-cap names may be used in the future for traits or properties with semantics dictated by the implementation, so never use an all-cap name for a property or trait.

3.8.3 Other future names

Any keywords or built-in operators introduced in future versions of Perl 6, other than all-cap words, will provide backward compatibility with existing programs by enabling the keyword’s special meaning only if the source text is marked (in that lexical scope) as being of the proper version.

Also need to make a statement about methods in standard classes and roles.

3.9 Keywords

Keywords are not “reserved words” as in some languages. They are terminals in the Perl 6 grammar, or are recognized as having special semantics. Obviously, variables having sigils will not conflict with keywords, and this offers some protection against the introduction of more keywords in the future. But identifiers used without sigils, such as type names and function calls, could potentially be in conflict. The recognition of a keyword is in context, so other uses of the name might or might not be a problem, depending on the situation.

An implementation may offer a feature to warn at compile-time about the use of a keyword as an identifier, but in general this is allowed and legal where the grammar is not looking for that keyword.

3.9.1 Table of keywords

any

also

begin

BEGIN
but
CALLER
CATCH
check
CHECK
class
CLASS
COMPILING
CONTEXT
CONTROL
copy
default
does
end
END
enum
enter
ENTER
export
first
FIRST
GLOBAL
handles
has
hidden
hides
init
INIT
is
keep
KEEP

last
LAST
leave
LEAVE
let
m
macro
method
multi
my
MY
next
NEXT
nextsame
of
only
our
OUR
OUTER
pre
PRE
post
POST
PROCESS
proto
q
ref
regex
redfined
ro (added by JMD)
role
rule

rw

rx

s

self

sub

submethod

subset

start

START

SUPER

temp

token

tr

undo

UNDO

use

VAR

where

3.9.2 Reserved trait names

The following are reserved as trait names. When used where traits are expected and with the proper trait verb, the built-in meaning is taken and any other use of that name is ignored.

For example, if you make a class named copy, you could not derive from it by coding "is copy" because that is reserved. However, coding "is ::copy" would take your class named copy.

is copy

is inline

is of

is ref

is ro

is rw

3.10 Operators and Punctuators

3.11 Literals

3.11.1 Integer

3.11.2 Floating Point

You may not write a Num as 42. with just a trailing dot. You must instead say either 42 or 42.0. In other words, a dot following a number can only be a radix point if the following character is a digit. Otherwise the postfix dot will be taken to be the start of some kind of method call syntax. (The .123 form with a leading dot is still allowed however when a term is expected, and is equivalent to 0.123 rather than $_.123.)

3.11.3 Infinities

```
Inf
-Inf
```

The name Inf is a built-in term that produces a value of type Infinite. This can be negated, as -Inf or any application of prefix:<-> on an expression that evaluates to the Inf value.

The values Inf and -Inf are distinct values that may be used with many other types. They have their own Infinite type, so the MMD system can dispatch on uses of Inf as distinct forms. For example

```
my @words = whatever;
my $min = Inf;
for @words {
  $min = $_  if $_ lt $min :lc;
  }
say "The first word is $min."
```

Even though the Str type doesn't have a concept of infinities like numeric classes do, the case-folding string comparison takes an Inf value as larger than any Str value. That gives a handy way to initialize the $min to a guaranteed over-large value, without having a special case inside the loop.

For more information, see the Infinite class on page 121.

3.11.4 Quoted Strings

For all quoting constructs that use user-selected brackets (see page 20), you can open with multiple identical bracket characters, which must be closed by the same number of closing brackets. Counting of nested brackets applies only to pairs of brackets of the same length as the opening brackets:

```
say #{{
   This comment contains unmatched } and { { { {          (ignored)
   Plus a nested {{ ... }} pair                           (counted)
}} q<< <<woot>> >>   # says " <<woot>> "
```

Note however that bare circumfix or postcircumfix `<<...>>` is not a user-selected bracket, but the ASCII variant of the `«...»` interpolating word list. Only `#` and the `q`-style quoters (including `m`, `s`, `tr`, and `rx`) enable subsequent user-selected brackets.

4 Basic Concepts

4.1 Translation and Execution

Perl 6 mixes up the compiling and execution stages, compared with other languages. With the explicitly supported ability to store a "compiled" program and copy it to other environments, we need to define this precisely.

4.2 Episodes

Some languages need to describe "phases of translation", and also have special periods during execution such as "before main", "during stack unwinding", etc.

Perl 6 interweaves translation and execution, so both sets of concepts exist side-by-side.

This specification defines specific periods of activity or states of the translator as *episodes*.

Episodes are nested.

In the descriptions below, the "first" and "last" non-episodic statement of a block refer not necessarily to the lexical ordering of statements in the block, but to the temporal ordering. That is the first one that gets executed when execution is transferred in, and the last one that gets executed before control was transferred out.

As explained in detail under *Episodic blocks* on page 62, statements can be lexically within a block surrounded by other statements, but temporally executed in a different episode than the non-episodic statements. These execute within the specified episode, and have specific semantic relationships with the lexically enclosed block.

4.2.1 Exception handling notes

If a block has a CATCH block attached to it, the handler is "installed" immediately before the ENTER episode and revoked immediately following the LEAVE episode. An exception thrown when the handler is not installed will behave as if the CATCH block were not part of the program source.

4.2.2 Context of Execution

If a closure created in the lexical scope of a block is executed in episodes of that block other than the normal execution episode, that closure may have trouble with lexical variables declared within that block.

Depending on the scoping declaration of the variable, the code that executes during an episode might have problems with those variables.

If code, executing during an episode, tries to access a variable that has not been set up for use yet, an exception is thrown. An implementation is free to diagnose this at compile-time and issue an error.

If the variable is subject to closure cloning, then accessing it before the START episode or during or after the LEAVE episode is an error.

If the variable is not subject to closure cloning, then accessing it before it is initialized (different scoping declarations cause initialization in different episodes, individually documented) is an error. The container does not exist, and is not even undef. It is outside the capability of the implementation to cope with.

Binding another variable to an uninitialized container may generate an error at that time, or may alternatively set the target variable to a similar "no container" state and delay error detection until that target variable is actually used.

4.2.3 Introspection

We need a standard function to find out what is the current episode. Note that finding out if you are handling an exception is a special case of this.

4.2.4 Summary

episodes drawing goes here.

4.2.5

4.2.6 Parsing episode

This is similar to "read time" in LISP. When the implementation is running the grammar to parse source text and build the parse tree, it is said to be in a parsing episode.

Your code can execute during a parsing episode if you modify the grammar.

4.2.7 blah blah

In order to properly define the characteristics of Macros, more episodes are needed, along with specific guarantees of how these stages take place.

4.2.8 BEGIN episode

Code that is executed immediately after it is compiled is in a BEGIN episode.

Variables that have a scoping declaration of constant are elaborated and initialized during a BEGIN episode.

During the BEGIN episode, all lexical variables that are affected by cloning will be in an indeterminate state: they may already be bound to the locations that will be used by the current execution of that closure, or they may be unbound.

4.2.9 CHECK episode

After a translation unit is parsed and translated, and before the first non-episodic statement or the INIT episode is executed, is the CHECK episode. This runs code and performs activities that are defined to take place during this episode.

If a translation unit is compiled but not executed[6], the CHECK still takes place as part of the call to parse and translate, and the INIT does not.

During the CHECK episode, all lexical variables that are affected by cloning will be in an indeterminate state: they may already be bound to the locations that will be used by the current execution of that closure, or they may be unbound.

4.2.10 INIT episode

When a Block object is executed for the first time, the INIT episode is performed before executing the first non-episodic statement in that Block. The INIT episode is performed before arguments are bound to the block's parameters, so code executed during the INIT episode will see them as not yet elaborated.

A Block object is only executed "for the first time" once, even if it is a closure that gets cloned. During the INIT episode, all lexical variables that are affected by cloning

6 Presumably the resulting Code object is stored for later use. Or, the CHECK itself can have side-effects and the Code object never used again!

will be in an indeterminate state: they may already be bound to the locations that will be used by the current execution of that closure, or they may be unbound.

4.2.11 START episode

When a BLOCK object is executed for the first time, or a fresh clone of a closure is executed for the first time, the START episode is performed. If the INIT episode is also performed, it will occur before the START episode.

If a BLOCK object is a subject to cloning, the lexical variables that are affected by cloning will be bound to the locations that will be used by the current execution of that closure.

Variables declared using state have their containers created when a closure is cloned, and are initialized in the START episode.

4.2.12 FIRST episode

Blocks that are executed as the body of a looping construct perform a FIRST episode the first time the loop is executed, each time the looping statement as a whole is executed. The FIRST episode is performed after the START episode (if applicable), and before the ENTER episode.

> To facilitate writing new constructs that behave like the built-in loops, there should be a standard way of triggering FIRST and LAST. The others all take care of themselves. But you need to tell the call that the FIRST is applicable this time (likewise for the last). Is there already a special form to invoke a Code object that is more general than just using postcircumscript:<( )>?

4.2.13 PRE episode

The PRE episode is performed before executing the first non-episodic statement of a block, and is specifically meant to assert preconditions associated with that block. If a precondition fails, the block containing it fails (*which ex?*) and subsequent processing of the block (e.g. the ENTER episode) is not performed.

4.2.14 ENTER episode

The ENTER episode is performed before executing the first non-episodic statement of a block. It is performed after the START episode (if applicable), and after parameters to the block have been bound to their arguments, after any PRE episode, and after any FIRST episode (if applicable).

Note that loops execute the entire block under their control, including ENTER and LEAVE episodes on every iteration of the loop.

The code executing in the ENTER episode will see that any exception handlers associated with the block have been registered into the dynamic scope; execution

before this episode will not.

4.2.15 normal execution episode

Non-episodic statements are executed in the "normal execution" episode.

Variables declared using my have their containers created and their value initialized when execution reaches the declaration statement.

> The following needs input from an expert. When are 'our' variables created and initialized? What episode does the pseudo-assignment run in (INIT? does that work?) It does say that it is different from Perl 5.

XXXX Item variables (using the $ sigil) declared using our have their containers created in the BEGIN episode and their value initialized when execution reaches the declaration statement for the first time. If the container is accessed before the value is initialized, it sees the default initialization value for non-compact types and uninitialized memory for compact types.

XXXX Other variables declared using our are unbound before the declaration statement is executed for the first time.

4.2.16 NEXT episode

For a block that is executed as the body of a looping construct, this is executed after the last non-episodic statement, but before the LEAVE episode. This is performed every time the block is executed as the body of a looping construct.

4.2.17 CATCH episode

If an exception is caught by a CATCH handler in the block, the body of the handler is performed in a CATCH episode. Transferring execution out of the main execution of a block or another episode will skip any intervening episodes: the CATCH episode is always followed by LEAVE or CONTROL on the same block.

The code in the CATCH episode will see any exception handlers for the block have already been revoked from the dynamic scope.

4.2.18 CONTROL episode

Transferring control out of an inner temporal block to an outer block is done by using CONTROL exceptions. This is an internal detail of the implementation, to catch the non-local transfer and jump to the correct statement within this block. Semantically, it works the same as CATCH episodes. This routing logic is performed in a CONTROL episode.

The CONTROL episode may follow a CATCH episode on the same block, if the exception handler issues a goto back to a non-episodic statement.

However, exceptions that occur during the CONTROL episode will not see the CATCH

handlers for the same block.

4.2.19 LEAVE episode

After executing the last non-episodic statement of a block, the LEAVE episode is performed. The code in the LEAVE episode will see any exception handlers for the block have already been revoked from the dynamic scope.

The topic will be set to the Capture that is to be returned from the function, and is rw so you may modify the return value from this code.

When are the KEEP and UNDO called—before or after this code? Or based on position of where they were added? Yes, single list ... it just skips the "wrong" ones. How can code sense if the parent block is failing or succeeding?

Is this when temp variables are restored?

4.2.20 KEEP episode

This is performed nested within the LEAVE episode, on blocks that are "successful".

A "successful" block is ...

4.2.21 UNDO episode

This is performed nested within the LEAVE episode, on blocks that are "unsuccessful".

An "unsuccessful" block is...

4.2.22 LAST episode

For a block that is executed as the body of a looping construct, the LAST episode is performed after the last iteration of the body for that execution of the looping statement as a whole. It is performed after the LEAVE episode.

Is it still performed if the body is executed zero times?

4.2.23 POST episode

The POST episode is performed after executing the last non-episodic statement of a block, and after the LEAVE episode. It is specifically meant to assert postconditions associated with that block. If a postcondition fails, an exception is thrown *(which)*.

The exception is thrown out of the current block, and will not be affected by a CATCH handler within this block, since the handler has already been revoked. The POST episode is performed after CATCH has been performed, if an exception is caught in the block.

The exception is thrown out of the current block, and will not be affected by a CATCH

handler within this block, since the handler has already been revoked. The POST episode is performed after CATCH has been performed, if an exception is caught in the block.

The topic will be set to the Capture that is to be returned from the function. It may be inspected (and postconditions coded for the return value) but not modified.

4.2.24 END episode

The END episode affects the first nested set of episodes only. That is, the main program loads modules which have their own sets of episodes, and modules have BEGIN and other episodic blocks that may invoke further nesting of the whole process. But the END applies to the outermost nested layer only[7].

The END episode takes place after the last non-episodic statement has executed, after the LEAVE episode, after the POST episode. The END episode is only ever followed by termination of the program in the Perl 6 implementation.

The behavior is as if there is a single global list of END blocks, and they are removed and executed in reverse order during the single END episode. Every block will append its END blocks to the global list during that block's INIT episode. If a block containing END blocks is executed for the first time during the END episode, they will be seen on the global list on the next iteration; the last one added will be the next one executed.

During the END episode, all lexical variables that are affected by cloning will be in an indeterminate state: they may already be bound to the locations that were used by some (first, last, or other) execution of the closure, or they may be unbound.

4.3 Abstract Syntax Tree

This translation stage needs to be standard and documented so that macros can be portable.

4.4 Start and Termination

4.5 Memory Model

Abstracted from physical RAM, garbage collected, but needs to address the raw-storage access idea for defining side-effects and program's behavior.

Define relationship of locations (will that be a formal term?), containers, and values. Illustrate closures, cloning, and aliasing. Cover read-only symbols.

7 There may be many END *episodic blocks* in the program, but they are executed in a single END episode when the program finishes.

4.6 Scope

4.6.1 Package-qualified names

You can use :: in several different ways.

... show them all, mark with C and R.

4.6.1.1 Compile-time resolution

The forms marked as “C” in the table above are guaranteed to resolve at compile time, exactly like a normal variable reference.

4.6.1.2 Run-time resolution

The forms marked as “R” in the table above resolve at run-time. This means that it is not an error if the symbol is not present at compile time, because the symbol may be added later.

Even if the optimizer folds constants or otherwise figures out what name you mean at compile time, it must behave as if it works at run time. Specifically, the behavior of this construct in closures must not change between different standard implementations just because one optimizes better. Worse, we don't want the meaning to change between debug and release builds of a program!

So even if an optimizer can save the run-time cost of searching, it should still check at run-time that the target scope still exists and issue an error that the symbol could not be found at run-time.

4.6.2 Pseudo-packages

```
MY          # Lexical variables declared in the current scope
OUR         # Package variables declared in the current package
GLOBAL      # Builtin variables and functions
PROCESS     # process-related globals
OUTER       # Lexical variables declared in the outer scope
CALLER      # Contextual variables in the immediate caller's scope
CONTEXT     # Contextual variables in any context's scope
SUPER       # Package variables declared in inherited classes
COMPILING   # Lexical variables in the scope being compiled
```

`GLOBAL::<$varname>` specifies the `$varname` declared in the `*` namespace. Or maybe it's the other way around...

`CALLER::<$varname>` specifies the `$varname` visible in the dynamic scope from which the current block/closure/subroutine was called, provided that variable is declared with the `"is context"` trait. (Implicit lexicals such as `$_` are automatically assumed to be contextual.)

`CONTEXT::<$varname>` specifies the `$varname` visible in the innermost dynamic scope that declares the variable with the `"is context"` trait.

`MY::<$varname>` specifies the lexical `$varname` declared in the current lexical scope.

`OUR::<$varname>` specifies the `$varname` declared in the current package's namespace.

`COMPILING::<$varname>` specifies the `$varname` declared (or about to be declared) in the lexical scope currently being compiled.

`OUTER::<$varname>` specifies the `$varname` declared in the lexical scope surrounding the current lexical scope (i.e. the scope in which the current block was defined).

We need two different kinds of OUTER. One binds at compile time, the other at run-time (which can be someone else's compile time). Consider use in default arguments.

Also look at SUPER with multiple base classes. Is there a way to refer uniformly to base classes without knowing the names, or to look at a flattened view of what the base classes bring (which may have conflicts). Methods are seen as subs in "package scope", right?

My own additions:

::LIKE::<$varname>

Like the "anchored types" in Eiffel, which uses the like keyword in that language. This returns the type object as a type. The sigil in front is needed because the inner one is wrong. OTOH, this doesn't work with the general mechanism, though the syntax works in this form. Needs a better idea.

The syntax $varname.WHAT returns the class object. You can declare it first:

my ::Typename ::= $varname.WHAT;

to capture a type that the library didn't provide an accessible name for. How can you use it directly in a declaration? Can :: be used as a prefix sigil operator?

::$varname.WHAT

or even just

::$varname

But ::($varname) already means something else. But I think that is only used after something else, including $::($varname) if it's first. And a sigil followed by a bracket is always special, so that's fine.

I think ::$varname should mean "like $varname" ala Eiffel.

IDLE::<$varname>

Doesn't do anything. Allows any variable to be "escaped" as a string, and makes unqualified names into qualified names.

4.7 Name Lookup

4.8 Sigils ($, @, %, etc.)

Perl 6 includes a system of **sigils** to mark the fundamental structural type of a variable:

```
$   scalar (object)
@   ordered array
%   unordered hash (associative array)
&   code/rule/token/regex
::  package/module/class/role/subset/enum/type/grammar
@@  slice view of @
```

Keep this section for just the basics. Move other stuff to relevant sections, on identifiers, terms, declarations, etc.

Within a declaration, the `&` sigil also declares the visibility of the subroutine name without the sigil within the scope of the declaration:

```
my &func := sub { say "Hi" };
func;   # calls &func
```

Within a signature or other declaration, the `::` sigil followed by an identifier marks a type variable that also declares the visibility of a package/type name without the sigil within the scope of the declaration. The first such declaration within a scope is assumed to be an unbound type, and takes the actual type of its associated argument. With subsequent declarations in the same scope the use of the sigil is optional, since the bare type name is also declared.

A declaration nested within must not use the sigil if it wishes to refer to the same type, since the inner declaration would rebind the type. (Note that the signature of a pointy block counts as part of the inner block, not the outer block.)

Sigils indicate overall interface, not the exact type of the bound object. Different sigils imply different minimal abilities.

`$x` may be bound to any object, including any object that can be bound to any other sigil. Such a scalar variable is always treated as a singular item in any kind of list context, regardless of whether the object is essentially composite or unitary. It will not automatically dereference to its contents unless placed explicitly in some kind of dereferencing context. In particular, when interpolating into list context, `$x` never

expands its object to anything other than the object itself as a single item, even if the object is a container object containing multiple items.

`@x` may be bound to an object of the `Array` class, but it may also be bound to any object that does the `Positional` role, such as a `List`, `Seq`, `Range`, `Buf`, or `Capture`. The `Positional` role implies the ability to support `postcircumfix:<[ ]>`.

Likewise, `%x` may be bound to any object that does the `Associative` role, such as `Pair`, `Mapping`, `Set`, `Bag`, `KeyHash`, or `Capture`. The `Associative` role implies the ability to support `postcircumfix:<{ }>`.

`&x` may be bound to any object that does the `Callable` role, such as any `Block` or `Routine`. The `Callable` role implies the ability to support `postcircumfix:<( )>`.

`::x` may be bound to any object that does the `Abstraction` role, such as a typename, package, module, class, role, grammar, or any other protoobject with `.HOW` hooks. This `Abstraction` role implies the ability to do various symbol table and/or typological manipulations which may or may not be supported by any given abstraction. Mostly though it just means that you want to give some abstraction an official name that you can then use later in the compilation without any sigil.

In any case, the minimal container role implied by the sigil is checked at binding time at the latest, and may fail earlier (such as at compile time) if a semantic error can be detected sooner. If you wish to bind an object that doesn't yet do the appropriate role, you must either stick with the generic `$` sigil, or mix in the appropriate role before binding to a more specific sigil.

An object is allowed to support both `Positional` and `Associative`. An object that does not support `Positional` may not be bound directly to `@x`. However, any construct such as `%x` that can interpolate the contents of such an object into list context can automatically construct a list value that may then be bound to an array variable. Subscripting such a list does not imply subscripting back into the original object.

Sigils are now invariant. `$` always means a scalar variable, `@` an array variable, and `%` a hash variable, even when subscripting. In item context, variables such as `@array` and `%hash` simply return themselves as `Array` and `Hash` objects. (Item context was formerly known as scalar context, but we now reserve the "scalar" notion for talking about variables rather than contexts, much as arrays are disassociated from list context.)

4.9 Lvalues

4.10 Closures

5 Other Conceptual Notes

Rather than spend too much time sticking on something, I'll drop it here for miscellaneous, to be filed later.

5.1 One-pass parsing

Informative

To the extent allowed by sublanguages' parsers, Perl is parsed using a one-pass, predictive parser. That is, lookahead of more than one "longest token" is discouraged. The currently known exceptions to this are where the parser must:

Locate the end of interpolated expressions that begin with a sigil and might or might not end with brackets.

Recognize that a reduce operator is not really beginning a [...] composer.

5.2 dots

A consequence of the postfix rule is that (except when delimiting a quote or terminating an unspace) a dot with whitespace in front of it is always considered a method call on `$_` where a term is expected. If a term is not expected at this point, it is a syntax error. (Unless, of course, there is an infix operator of that name beginning with dot. You could, for instance, define a Fortranly `infix:<.EQ.>` if the fit took you. But you'll have to be sure to always put whitespace in front of it, or it would be interpreted as a postfix method call instead.)

For example,

```
foo .method
```

and

```
foo
.method
```

will always be interpreted as

```
foo $_.method
```

but never as

```
foo.method
```

Use some variant of

```
foo\
.method
```

if you mean the postfix method call.

One consequence of all this is that you may no longer write a Num as `42.` with just a trailing dot. You must instead say either `42` or `42.0`. In other words, a dot following a number can only be a decimal point if the following character is a digit. Otherwise the postfix dot will be taken to be the start of some kind of method call syntax, whether long-dotty or not. (The `.123` form with a leading dot is still allowed however when a term is expected, and is equivalent to `0.123` rather than `$_.123`.)

5.3 Undefined types

Copied from S02, needs work.

These can behave as values or objects of any class, except that `defined` always returns false. One can create them with the built-in `undef` and `fail` functions. (See S04 for how failures are handled.)

```
Object      Uninitialized (derivatives serve as protoobjects of classes)
Whatever    Wildcard (like undef, but subject to do-what-I-mean via MMD)
Failure     Failure (lazy exceptions, thrown if not handled properly)
```

Whenever you declare any kind of type, class, module, or package, you're automatically declaring a undefined prototype value with the same name.

I don't understand the following.

Use `Object` for the most generic non-failure undefined value. The `Any` type is also undefined, but excludes `Junctions` so that autothreading may be dispatched using normal multiple dispatch rules.)

5.4 Even more miscellaneous

What is “rw”? It uses the syntax of a trait, but needs to be more magical than just a role.

Numeric values in untyped variables use `Int` and `Num` semantics rather than `int` and `num`.

6 Execution Context

In string contexts, container objects automatically stringify to appropriate (white-space separated) string values. In numeric contexts, the number of elements in the container is returned. In boolean contexts, a true value is returned if and only if there are any elements in the container.

7 Terms

7.1 Variables and Sigils

7.1.1 Code Objects

Unlike in Perl 5, the notation `&foo` merely stands for the `foo` function as a Code object without calling it. You may call any Code object with parens after it (which may, of course, contain arguments):

```
&foo($arg1, $arg2);
```

Whitespace is not allowed before the parens because it is parsed as a postfix. As with any postfix, there is also a corresponding `.()` operator, and you may use the "unspace" form to insert optional whitespace and comments between the backslash and either of the postfix forms:

```
&foo\   ($arg1, $arg2);
&foo\   .($arg1, $arg2);
&foo\#[
    embedded comment
].($arg1, $arg2);
```

With multiple dispatch, `&foo` may not be sufficient to uniquely name a specific function. In that case, the type may be refined by using a signature literal as a postfix operator:

```
&foo:(Int,Num)
```

It still just returns a `Code` object. A call may also be partially applied by using the `.assuming` method:

```
&foo.assuming(1,2,3,:mice<blind>)
```

7.2 * "Whatever"

The * symbol by itself, when used as a term, refers to a constant sometimes called "whatever". It is defined something like this:

```
sub term:<*> ()
of Whatever
is lvalue
is rw
 {
 class ItemDumper is Tie::Scalar {
```

```
    sub STORE ($x) {
      # throw it away.  Eats assignments.
      }
    # ... more stuff to make it work right
    };
  state Whatever $dump is ItemDumper = Whatever;
  return $dump;
  }
sub term:<**> ()
of Whatever
is inline
 {
 state Whatever $x= Whatever{ :multidimensional };
 return $x;
 }
```

That is, using the * or ** as a term produces an undef value that has type of Whatever. See page 143 for more about the Whatever type. Various functions and operators respond to an argument of type Whatever in an appropriate way.

For example:

```
if $x ~~ 1..* {...}              # if 1 <= $x <= +Inf
my ($a,$b,$c) = "foo" xx *;      # an arbitrary long list of "foo"
if /foo/ ff * {...}              # a latching flipflop
@slice = @x[*;0;*];              # any Int
@slice = %x{*;'foo'};            # any keys in domain of 1st dimension
@array[*]                        # flattens, unlike @array[]
```

The term * also behaves as a lvalue in an unusual way: it ignores assignment to it.

```
(*, *, $x) = (1, 2, 3);          # skip first two elements
```

This replaces the feature of assigning to undef from Perl 5. The ** term does not have this behavior and is not an lvalue.

The optimizer is free to understand the special semantics of the container produced by the * term, as well as the semantics of the various built-in functions that understand a Whatever argument.

The ** form means “multidimensional”. Example functions that understand ** are XXXXX give examples XXXXX.

7.3 Captures

An argument list may be captured into an object with backslashed parens:

```
$args = \(1,2,3,:mice<blind>)
```

Values in a `Capture` object are parsed as ordinary expressions, marked as invocant, positional, named, and so on.

Like `List` objects, `Capture` objects are immutable in the abstract, but evaluate their arguments lazily. Before everything inside a `Capture` is fully evaluated (which happens at compile time when all the arguments are constants), the eventual value may well be unknown. All we know is that we have the promise to make the bits of it immutable as they become known.

`Capture` objects may contain multiple unresolved iterators such as feeds or slices. How these are resolved depends on what they are eventually bound to. Some bindings are sensitive to multiple dimensions while others are not.

You may retrieve parts from a `Capture` object with a prefix sigil operator:

```
$args = \3;      # same as "$args = \(3)"
$$args;          # same as "$args as Scalar" or "Scalar($args)"
@$args;          # same as "$args as Array"  or "Array($args)"
%$args;          # same as "$args as Hash"   or "Hash($args)"
```

When cast into an array, you can access all the positional arguments; into a hash, all named arguments; into a scalar, its invocant.

All prefix sigil operators accept one positional argument, evaluated in item context as a rvalue. They can interpolate in strings if called with parentheses. The special syntax form `$()` translates into `$( $/ )` to operate on the current match object; the same applies to `@()` and `%()`.

What is the above paragraph doing here?

`Capture` objects fill the ecological niche of references in Perl 6. You can think of them as "fat" references, that is, references that can capture not only the current identity of a single object, but also the relative identities of several related objects. Conversely, you can think of Perl 5 references as a degenerate form of `Capture` when you want to refer only to a single item.

7.4 Signatures

A signature object (`Signature`) may be created with colon-prefixed parens:

```
my ::MySig ::= :(Int, Num, Complex, Status)
```

Expressions inside the signature are parsed as parameter declarations rather than ordinary expressions. See S06 for more details on the syntax for parameters.

Signature objects bound to type variables (as in the example above) may be used within other signatures to apply additional type constraints. When applied to a `Capture` argument, the signature allows you to take the types of the capture's arguments from `MySig`, but declare the (untyped) variable names yourself via an additional signature in parentheses:

```
sub foo (Num Dog|Cat $numdog, MySig $a ($i,$j,$k,$mousestatus)) {...}
```

```
    foo($mynumdog, \(1, 2.7182818, 1.0i, statmouse());
```

8 Expressions

8.1 Postfix and Postcurmfix

Subscripts now consistently dereference the container produced by whatever was to their left. Whitespace is not allowed between a variable name and its subscript. However, there are two ways to stretch the construct out visually. Since a subscript is a kind of postfix operator, there is a corresponding **dot** form of each subscript (`@foo.[1]` and `%bar.{'a'}`) that makes the dereference a little more explicit. Constant string subscripts may be placed in angles, so `%bar.{'a'}` may also be written as `%bar<a>` or `%bar.<a>`. Additionally, you may insert extra whitespace using the unspace.

8.2 Subscripting

Applies to Arrays and Hashes. Be sure to distinguish if something differs.

Slicing is specified by the nature of the subscript, not by the sigil.

Subscripts are always evaluated in list context.

8.2.1 Types of Braces

8.2.2 Item or Slice return

Need to explain how the return value knows to be item or list context!!

8.2.3 Multidimensional Subscripts

9 Statements

10 Declarations

Details of syntax...

10.1 Binding

Use of := and ::= when declaring a new symbol name.

Works the same as parameter passing (we hope).

10.2 Hierarchical types

A non-scalar type may be qualified, in order to specify what type of value each of its elements stores:

```
my Egg $cup;                        # the value is an Egg
my Egg @carton;                     # each elem is an Egg
my Array of Egg @box;               # each elem is an array of Eggs
my Array of Array of Egg @crate;    # each elem is an array of arrays of Eggs
my Hash of Array of Recipe %book;   # each value is a hash of arrays of Recipes
```

Each successive `of` makes the type on its right a parameter of the type on its left. Parametric types are named using square brackets, so:

```
my Hash of Array of Recipe %book;
```

actually means:

```
my Hash[of => Array[of => Recipe]] %book;
```

Because the actual variable can be hard to find when complex types are specified, there is a postfix form as well:

```
my Hash of Array of Recipe %book;          # HoHoAoRecipe
my %book of Hash of Array of Recipe;       # same thing
```

10.3 Hashes

Things specific to declaring hashes, including how to specify the key type, multiple dimensions.

10.4 Pseudo-assignment

In a declaration, the = or .= symbol is used in a manner that looks like assignment, but is not. Declaring a variable actually uses a Signature as the syntactic unit, so initializers to variables are syntactically the same as default values to parameters. The initializer is parsed and stored as a closure as part of the Signature object.

If the Signature is used to declare variables, this closure run immediately (for my

variables) or is bound to a trait of the variable.

Which trait used depends on the scoping declarator, and reflects the natural lifetime of the container.

```
my $foo = 1;          # happens at the same time as normal assignment
our $foo = 1          # happens at INIT time
has $foo = 1;         # happens at BUILD time
state $foo = 1;       # happens at START time
constant $foo = 1;    # happens at BEGIN time
constant foo = 1;     # happens at BEGIN time
```

I recall some discussion of what extra braces do. But I can't find it now. Is it sometimes automatically transformed differently, or was I dreaming?

For consistency, pseudo-assignment always parses the argument as a closure, even if it happens at the same time as normal assignment. That is, changing the scoping declarator will not change how the initializer is parsed, just when it is scheduled to be executed. (*proposal*)

10.5 Variable Initialization

A variable declaration may contain an initializer, which is written using = or .=, and is a pseudo-assignment.

Syntactically, variable declaration is done with a Signature (see page 85), and the initializer is the default value of the variable in the signature.

10.5.1 * ("whatever") initialization

A variable initialization of the * token, when the variable has a fully-specified value type that is a compact type, is taken to mean "no initialization" and disables the default initialization. The bits in the compact type's storage will be whatever they were when allocated from lower-level implementation details, and no extra work will be performed to put them into a known state.

For non-compact types or incompletely-specified types, the * is an error.

10.5.2 Default Initialization

If no initialization is specified, then the default initialization is performed.

1.1.1.1 Item Variables

For an item variable (declared using the $ sigil), the default initializer for a variable declaring fully-specified value type is the default undefined prototype for that type, when that type supports undefined prototypes:

```
my Dog $spot;         # $spot is initialized with ::Dog
my Dog $spot = Dog;   # same thing
```

For compact types, which do not have undefined prototypes, the default value is specified in this table:

int, uint, of all sizes	0
num of all sizes	NaN
bool	false (binary 0)
rat of all sizes	0/0
complex of all sizes	NaN + NaNi
buf of all sizes	binary zeros

For variables that have incompletely-specified types, the default initializer is the undifferentiated undef value.

1.1.1.2 Array and Hash Variables

For variables defined using the @ or % sigils, the default initializer is a new instance of the container type. The instance will be set up with details such as array size or key types that are specified, even if not as explicit parameters to a stated container type. The container instance may be lazy about setting up its internal state until values are actually stored in it, but even so it is still considered a defined (though empty) object of that container type.

The value type is given as a parameter to the container type, so when the container needs to create new slots for values that are not given at that time, it knows how to create default values in the same way as items.

```
Num @list;  # defined but empty Array object
Num @list = Array.new;  # same thing
@list[2] = 5;
# implies:
  my Num $blank;  # auto-default
  @list[0] = @list[1] = $blank;
```

10.6 Subroutines

10.6.1 Kinds of Subroutines

10.6.1.1 sub

10.6.1.2 method

10.6.1.3 submethod

10.6.1.4 operators

10.6.1.5 conversions

Conversions are functions inside a class that convert values of that class into other types. They are usually not called directly but are used when conversion is necessary, such as implicit coercion and when using a type name as a listop.

The syntax uses a grammatical category name like operators do:

```
method conversion:<Str> () { ... }
```

The "of" return type is not necessary. If specified, it must be the same as the conversion's named type.

It may be a method or a submethod.

It may not be multi. The only keyword is redundant but allowed.

As a method or submethod, it has an invocant parameter that may be omitted or declared in the parameter list in the usual manner. Additional arguments may be declared as well, in the usual manner. The additional arguments can be used to control details of the conversion. Typically these are adverbs.

The text in the category's parameter is parsed as a type name that must be matched at compile time. More specifically, it is handled in exactly the same way as a type would be in the parameter list, e.g.

```
method foo (Num $dummy, Int $decimalplaces?) { ... }
method conversion:<Num> (Int $decimalplaces?) { ... }

$p.foo(Num, 5);  # footnote [8]
$p.conversion:<Num>(5);
```

Consider how the Num used in the definition of foo is parsed and handled in classes and roles, including how it involves virtual class names. The Num used in conversion:<Num> is treated in the exact same way in all cases.

The conversion function may take the traits implicit or explicit.

```
method conversion:<Num> ()  is implicit { ... }
```

8 Num is called to produce an undefined protoobject. foo is ignoring the value, so I don't have to pass it anything in particular, and undef will do just fine if it's typed properly.

If implicit, the implementation may call this function implicitly to perform an implicit conversion. If declared either implicit or explicit, the implementation may call this function in response to using the conversion type as a listop.

```
Num $x;
$x = $p;  # OK only if declared as implicit.
$x = Num $p, 5;  # OK if implicit or explicit.
```

If neither is declared, explicit is assumed by default.

Note that the implicit/explicit behavior can be overridden with the use conversion pragma.

10.6.2 Parameters

Parameters may be given types, just like any other variable:

```
sub max (int @array is rw) {...}
sub max (@array of int is rw) {...}
```

10.6.2.1 Generic types

Within a declaration, a class variable (either by itself or following an existing type name) declares a new type name and takes its parametric value from the actual type of the parameter it is associated with. It declares the new type name in the same scope as the associated declaration.

```
sub max (Num ::X @array) {
    push @array, X.new();
}
```

The new type name is introduced immediately, so two such types in the same signature must unify compatibly if they have the same name:

```
sub compare (Any ::T $x, T $y) {
    return $x eqv $y;
}
```

10.6.3 Return type

The `as` form may be used in subroutines:

```
my sub get_book ($key) as Hash of Array of Recipe {...}
```

Alternately, the return type may be specified within the signature:

```
my sub get_book ($key --> Hash of Array of Recipe) {...}
```

There is a slight difference, insofar as the type inferencer will ignore a `as` but pay attention to `-->` or prefix type declarations, also known as the `of` type. Only the

inside of the subroutine pays attention to `as`, and essentially coerces the return value to the indicated type, just as if you'd coerced each return expression.

You may also specify the `of` type as the `of` trait (with `returns` allowed as a synonym):

```
my Hash of Array of Recipe sub get_book ($key) {...}
my sub get_book ($key) of Hash of Array of Recipe {...}
my sub get_book ($key) returns Hash of Array of Recipe {...}
```

On a scoped subroutine, a return type can be specified before or after the name. We call all return types "return types", but distinguish two kinds of return types, the `as` type and the `of` type, because the `of` type is normally an "official" named type and declares the official interface to the routine, while the `as` type is merely a constraint on what may be returned by the routine from the routine's point of view.

```
our sub lay as Egg {...}            # as type
our Egg sub lay {...}               # of type
our sub lay of Egg {...}            # of type
our sub lay (--> Egg) {...}         # of type

my sub hat as Rabbit {...}          # as type
my Rabbit sub hat {...}             # of type
my sub hat of Rabbit {...}          # of type
my sub hat (--> Rabbit) {...}       # of type
```

If a subroutine is not explicitly scoped, it belongs to the current namespace (module, class, grammar, or package), as if it's scoped with the `our` scope modifier. Any return type must go after the name:

```
sub lay as Egg {...}                # as type
sub lay of Egg {...}                # of type
sub lay (--> Egg) {...}             # of type
```

On an anonymous subroutine, any return type can only go after the `sub` keyword:

```
$lay = sub as Egg {...};            # as type
$lay = sub of Egg {...};            # of type
$lay = sub (--> Egg) {...};         # of type
```

but you can use a scope modifier to introduce an `of` prefix type:

```
$lay = my Egg sub {...};            # of type
$hat = my Rabbit sub {...};         # of type
```

Because they are anonymous, you can change the `my` modifier to `our` without affecting the meaning.

The return type may also be specified after a `-->` token within the signature. This doesn't mean exactly the same thing as `as`. The `of` type is the "official" return type, and may therefore be used to do type inferencing outside the sub. The `as` type only makes the return type available to the internals of the sub so that the `return` statement can know its context, but outside the sub we don't know anything about

the return value, as if no return type had been declared. The prefix form specifies the `of` type rather than the `as` type, so the return type of

```
my Fish sub wanda ($x) { ... }
```

is known to return an object of type Fish, as if you'd said:

```
my sub wanda ($x --> Fish) { ... }
```

not as if you'd said

```
my sub wanda ($x) as Fish { ... }
```

It is possible for the `of` type to disagree with the `as` type:

```
my Squid sub wanda ($x) as Fish { ... }
```

or equivalently,

```
my sub wanda ($x --> Squid) as Fish { ... }
```

This is not lying to yourself--it's lying to the world. Having a different inner type is useful if you wish to hold your routine to a stricter standard than you let on to the outside world, for instance.

11 Closures

In Perl 6, every block is a closure. As objects, the terms are synonymous. However, the term "block" is used to emphasize the structure of the source text as a list of statements wrapped in curly braces; and the term "closure" is used to emphasize the fact that the compiled Code object refers to locations that were found using symbols in scopes that no longer exist.

It is also helpful to have more than one pronoun, so we may write, "the block containing the closure..." without making the legaleeze too dense.

11.1 in-place blocks

An in-place block is executed at the location in the program flow in which it is defined. That is, is appears as though the braces are used only for grouping.

Semantically, in-place blocks are closures that are called automatically at the point where they are defined. You can sense this is different from plain grouping of statements because CALLER refers to the surrounding block.

```
my $a = 5;
my $cl = { #define it
  my $sym1 := $CALLER::a;
  say $sym1 =:= $a;
  }
$cl();  # call it immediately.
{  my $sym2 := $CALLER::a;
  say $sym2 =:= $a;
  }
```

The second block has the same semantics as the first. It sees the variable $a, not only in its OUTER scope, but in its CALLER as well. This is also true of blocks used in looping and control structures. Blocks are always called, in the same way as subroutines.

A block is in-place if it is *bare* and used as a statement or interpolated into a string literal.

A block is *bare* when it does not introduced by the sub keyword or the pointy-block syntax.

A closure interpolated into a string literal is evaluated in string item context.

```
say "The attribute is { @baz[3](1,2,3){$xyz}<blurfl>.attr }."
```

11.2 Closures vs Routines

Closures, in general, are not Routines. All Routine declarations have an explicit *routine declarator* such as sub or method; bare blocks and pointy blocks are never routines.

See for example the behavior of return. (Page 108)

11.3 Literal Hashes

In Perl 6, braces are also used for constructing hashes. This is understood to be a closure that produces a hash as its return value.

```
my %hash = { a=>1, b=>2 }
```

This occurs when the block is bare and is either empty or contains a single list starting with a pair or a hash.

```
my $x = { a => 1, b=>2 }
my $y = { 2+2 }
```

In the above example, the overall syntax is the same. If you cover up the contents of the block, you cannot tell which is which. Only knowing that the first one contains a single list and starts with a pair, can you know that $x is being assigned a hash. $y, on the other hand, is being assigned a closure.

```
say $x<a>;  # use $x as a ref to a hash
say $y(); # use $y as a ref to a closure
```

11.4 Cloning

A closure may refer to symbols that exist when the closure is referenced at run-time, and are different when it is referenced again later.

```
sub tim (Num $x)
{
 my $y = $x * $x + 1;
 my $closure;
 $closure = { return $^param + $y }  # cloning takes place here
 return $closure;
 }

my $c1 = tim(4);
my $c2 = tim(3);
say $c1(1);  # prints 18
say $c2(1);  # prints 11
say $c1 =:= $c2;  # prints False
```

It is clear that the Code objects referred to by $c1 and $c2 are different. Calling them with the same parameters produces different results. That's because, although the "same" block of code was assigned to $closure each time, it refers to a different instance of $y. The stack frame containing $y only exists at run time, and each time tim is called, a different $y is created.

Producing a closure is a process that must be completed at run time. The code compiled for the block is incomplete, and must be "fixed up" with the proper meaning for $y.

This is called *cloning* the closure, and it happens at the point in execution where the closure is referenced. The right hand side of the assignment, which looks like it is just the block itself, is actually compiled to a function that takes the pre-compiled unchanging part of the block and binds in the dynamic run-time information that must be found at the proper time during execution, and returns a unique Closure object.

Such symbols that must be found at run-time are said to be "subject to cloning". The time of cloning is when the completed Closure object is first needed; that is, when it is used as a value, called, or bound to another symbol.

```
my $prev;
for ^10 -> $loop {
  my $x= $loop;
  my $closure;
  restart:
  $closure = { 1 + $x } # cloning takes place here
  say $closure =:= $prev;
  $prev = $closure;
  if $x == 1 {
    ++$x;
    goto restart;
    }
  } # end of for
```

Each time through the for loop, the block forming the body of the loop is entered afresh, and $x is a different symbol. So, each $closure is a different clone. This is true most of the time when the point of the closure is encountered more than once during execution.

However, if the goto is used, the same closure is referred to and it is not re-made. The assignment is redundant, as it gets the same Code object again. It is not cloned again.

If the block containing the closure is not exited and re-entered, the closure is only cloned when the closure is first needed. Cloning is only performed once during the lifetime of the scope immediately containing it.

However, the implementation can reuse Code objects and not re-clone when it can tell that they are the same. If the closure does not refer to any symbols subject to cloning, that is easy: it may optimize the process away and return the same Closure object every time. If the closure refers to symbols that are in outer blocks, but not its immediate enclosing block, then the implementation may detect that and optimize the cloning out of the loop, returning the same Closure object for a long as that outer block is not exited.

To summarize

1. The following symbols are subject to cloning:

- â variables declared as state.
- â symbols from outer scopes.

2. Cloning is performed when the closure is first needed during the lifetime of its immediate surrounding scope.
3. Cloning is not performed again for subsequent uses of the same closure within the lifetime of its immediate scope.
4. Cloning may be optimized out in cases where subsequent uses refer to the same symbols.

11.5 "closing" over the symbols

When a closure is cloned, it makes its own bindings to the various symbols in outer scopes that are referred to. Only those symbols that are referenced at compile-time are so treated.

All symbols that are subject to cloning are bound to anonymous symbols that are part of the Closure object, and not mentioned in any scope. Use of those symbols within the closure refer to those internal symbols.

Symbolic references that resolve at run-time may not have the original surrounding lexical scopes to look at. Even if a symbolic reference refers to a symbol that was subject to cloning, the closure uses its internal copy and does not maintain any association with where it originally was cloned from.

```
my $faraway;
sub outer ()
  {
  my $x;
  my $y;
  my $closure = {
    my $z = $x + 1;
    my $q1 = OUTER::('$x');
    my $q2 = OUTER::('$y');
    my $q3 = OUTER::OUTER::('$faraway');
    }
  return $closure;
  } # end outer

my $c = outer;
$c();
```

Note that the symbolic reference form is used that specifies run-time lookup of the symbol table. OUTER::('$x') is run-time, in contrast to OUTER::<$x> which would be performed at compile-time (see *Run-time resolution* on page38)

When the closure is eventually called, it has no trouble with $z because $x was

referred to at compile-time and that's what closures *do*.

It does have trouble with `$q2`, since the OUTER scope is no longer available. It also has trouble with `$q1`, even though it is trying to point to the same `$x` that was indeed "closed over" by the closure, it does not remember where it came from so just because it's `$x` doesn't matter: it cannot find it.

However, the symbolic reference in `$q3` does work. When it is executed, the scope in question still exists. At compile-time, it knows that OUTER::OUTER:: refers to the specific scope at the top of the listing, and notes what scope that is during cloning. When the symbolic reference is made at run-time, the dynamic scope is walked back looking for the particular one it needs. If that scope is found (is still alive on the stack), the symbolic reference proceeds. If not, as in the case of `$q1`, it fails.

In order for the target scope to me noted at cloning time, the scope itself must be mentioned as a symbol at compile time. So OUTER::OUTER::('$faraway') gets that treatment, but ::OUTER::('OUTER::$faraway') or ::('OUTER::OUTER::$faraway') does not.

This is true for any named block or outer scope, not just the pseudo-packages.

Referring to an outer scope, as a symbol, from a closure does not cause its lifetime to be extended in the same manner as normal variables and other symbols. Rather, it notes which scope that is, and at run-time it tries to find it again on the stack. If it is no longer present on the stack, it fails.

You may check the symbolic name of an outer scope (either named block or pseudo-package) for defined, and it returns false if the scope is not found. Attempting to reference something in the undefined scope throws an exception (*which?*)

Pseudo-packages that refer to temporal scopes are evaluated at run-time, in the context of the callers of the closure. Other pseudo-packages are evaluated at compile time to the scope that exists when the closure is compiled, and noted at cloning time for re-locating at run-time.

11.6 Episodic blocks

A number of traits are understood to have special meaning by the Perl 6 implementation.

name	**multiple**	**expr.**	**when called**
BEGIN	n/a	yes	at compile time, ASAP, only ever runs once
CHECK	list, n/a	yes	at compile time, ALAP, only ever runs once
INIT	list	yes	at run time, ASAP, only ever runs once
END	list	no	at run time, ALAP, only ever runs once
START	list	yes	on first ever execution, once per closure clone
ENTER	list	yes	at every block entry time, repeats on loop blocks.

LEAVE	list	no	at every block exit time
KEEP	list	no	at every successful block exit, part of LEAVE queue
UNDO	list	no	at every unsuccessful block exit, part of LEAVE queue
FIRST	list	yes	at loop initialization time, before any ENTER
NEXT	list	no	at loop continuation time, before any LEAVE
LAST	list	no	at loop termination time, after any LEAVE
PRE	list	no	assert precondition at every block entry, before ENTER
POST	list	no	assert postcondition at every block exit, after LEAVE
CATCH	item	no	catch exceptions, before LEAVE
CONTROL	item	no	catch control exceptions, before LEAVE

If these traits are defined on a closure, they are called at specific times automatically by the implementation.

11.6.1 Episodic blocks

Keywords for the episode names in the table above may be used to introduce a statement that is to be executed in the specified episode of the immediate enclosing block rather than in normal sequence.

The trait name is used as a keyword, followed by a closure. For example,

```
{  # block begins here
◦◦◦
PRE { $x > 0 }
INIT { say "I'm here!" }
◦◦◦
}
```

This has the effect of appending the closure { $x > 0 } to the array stored as the PRE trait of the block that starts at the beginning of the example. Likewise, the block { say "I'm here!" } is appended to the array stored as the INIT trait of the block.

A POST block declared immediately within a PRE or ENTER block are not applied as traits to that containing block, but are promoted to apply to the same block as that PRE or ENTER.

```
method push ($new_item) {
   ENTER {
      my $old_height = self.height;
      POST { self.height == $old_height + 1 }
```

```
    }

    $new_item ==> push @.items;
  }

 method pop () {
    ENTER {
      my $old_height = self.height;
      POST { self.height == $old_height - 1 }
    }

    return pop @.items;
 }
```

In the above example, the POST blocks define code to execute as the POST of the methods, not the POST of the ENTER blocks. Yet, defining them within the ENTER block gives access to the local variable defined there.

Clarification: If a POST block *p* appears inside another block *b2* that is itself inside a PRE or ENTER block *b1*, and b2 is not a PRE or ENTER block, no special rules are used, and the POST block has the effect of creating or appending to a trait on *b2*.

The FIRST, NEXT, and LAST blocks may only appear in the block that is the body of a looping statement. It must be the top-level block, not a block nested inside such a block.

> So, a closure that is passed as a argument to code that executes it more than once, and is meant to work just like the built-in looping constructs, can't use it? Such code could be programmed to specifically look for these traits and use them properly. There should be a way to mark the body as a loop, to enable this.

```
for @list -> $item {
  my $total;
  FIRST { $total = 0 }
  if ($item == 5) {
    FIRST { say "I'm here" }   # error
    }
  } # end of for loop
```

The first FIRST block causes $total = 0 to be executed only the first time through the loop.[9] The second FIRST block is a compile-time error. It would apply to the block that is the body of the if statement, and have nothing to do with the enclosing for loop. So to prevent mistakes, this is deemed to be a compile-time error.

9 The closures in the FIRST trait are cloned to go with this instance of the loop's execution, so it does refer to the proper instance of $total.

11.6.2

11.6.3 Semantics

Some of the trait's closures are not just called at some time, but require specific semantics.

PRE, POST, CATCH, LEAVE

> Need to define these details.

11.6.4 Expressions

Those special statement blocks marked as "yes" under "expr." in the table may be used as expressions and return values.

```
my $compiletime = BEGIN { localtime };
our $temphandle = START { maketemp() };
```

The return value is stored in a location implicitly created by the compiler. That location is thus assigned to when the closure is run at the prescribed time. The expression containing the special block form is then compiled to refer to that location. So the above is equivalent to:

```
constant $*nonce1 = localtime;
my $compiletime = $*nonce1;
state $*nonce2 = maketemp();
our $temphandle = $*nonce2;
```

The temporary location will be global or associated with each block instance as appropriate for the trait.

BEGIN, CHECK, INIT: only runs once, so only one location needed.

START, ENTER, FIRST: separate location for each instance of the block.

> Check example again after clarifying state/our issue noted earlier (page 35). Is this example correct?

11.6.5 Setting on variables

Some of these (*which?*) have lowercase equivalents that may be set as traits on variables. These register closures in their enclosing block, the same as the upper-case forms. But they execute with the variable as their topic.

```
my $r will start { .set_random_seed() };
our $h will enter { .rememberit() } will undo { .forgetit() };
```

is approximately equivalent to:

```
my $r;
START { $r.set_random_seed() }
our $h;
```

```
ENTER { $h.rememberit() }
UNDO { $h.forgetit() }
```

That doesn't completely work... is $r elaborated (defined) at START time? In the examples, what do the member functions do on undefined untyped values anyway? See S04 under "Closure traits"

The similarity between PRE block and pre traits of variables etc. extents to the special behavior of having a POST blocks inside PRE or ENTER blocks. A pre or post trait on a variable declared immediately inside an ENTER or PRE block will promote the location of the applied block trait in exactly the same manner as the corresponding episodic block.

11.6.6 Explicit use of traits

Given a reference to a closure (type Block), you can examine most of these values just like you could any other trait. You can modify their values too.

The exception is BEGIN blocks, which are executed immediately after parsing, and do not need to be stored anywhere. There is no BEGIN trait to examine. Similarly, CHECK may be omitted by the implementation because it only has effect when the block is being compiled. Code executed within BEGIN blocks, macros, or other code that is executing while the target block is still being compiled can see and modify the CHECK trait in the normal way. But after executing the CHECK trait when compilation of that block is completed, the implementation may delete it or otherwise make it unavailable as a trait.

11.6.7 Use in top-level code

Statements used in top-level code in a file, where there are no surrounding braces, are still compiled as part of a closure that holds the entire file or the implied body of the package et al.

12 The Type System

In support of OO encapsulation, there is a new fundamental datatype: **P6opaque**. External access to opaque objects is always through method calls, even for attributes.

Perl 6 has an optional type system that helps you write safer code that performs better. The compiler is free to infer what type information it can from the types you supply, but will not complain about missing type information unless you ask it to.

Types are officially compared using name equivalence rather than structural equivalence. However, we're rather liberal in what we consider a name. For example, the name includes the version and authority associated with the module defining the type (even if the type itself is "anonymous"). Beyond that, when you instantiate a parametric type, the arguments are considered part of the "long name" of the resulting type, so one `Array of Int` is equivalent to another `Array of Int`. (Another way to look at it is that the type instantiation "factory" is memoized.) Typename aliases are considered equivalent to the original type.
This name equivalence of parametric types extends only to parameters that can be considered immutable (or that at least can have an immutable snapshot taken of them). Two distinct classes are never considered equivalent even if they have the same attributes because classes are not considered immutable.

12.1 Junction Types

> This was called Polymorphic in S02, but I think that is a bad term. Polymorphism expresses the same interface in different ways. This expresses different interfaces at different times.

Anywhere you can use a single type you can use a set of types, for convenience specifiable as if it were an "or" junction:

```
my Int|Str $error = $val;              # can assign if $val~~Int or $val~~Str
```

Fancier type constraints may be expressed through a subtype:

```
subset Shinola of Any where {.does(DessertWax) and .does(FloorTopping)};
if $shimmer ~~ Shinola {...}  # $shimmer must do both interfaces
```

Since the terms in a parameter could be viewed as a set of constraints that are implicitly "anded" together (the variable itself supplies type constraints, and `where` clauses or tree matching just add more constraints), we relax this to allow juxtaposition of types to act like an "and" junction:

```
# Anything assigned to the variable $mitsy must conform
# to the type Fish and either the Cat or Dog type...
my Cat|Dog Fish $mitsy = new Fish but { int rand 2 ?? .does Cat
                                                   !! .does Dog };
```

Example needs work. .does Cat isn't going to add a trait but tests for the existence of the role, right?

12.2 Typing with Variables, Containers, and Values

The value type of an entity (a variable, or a return value from a function call) is a constraint indicating what sorts of values the variable may contain. More precisely, it's a promise that the object or objects contained in the variable are capable of responding to the methods of the indicated static type.

A static type may be a simple type name or role name, which indicates that a value so-marked is known to support the stated interface.

In the case of subtypes, the values must satisfy the constraints on the subtype. This may be used for additional assumptions by the compiler to the extent that it understands the constraint.

A static type may also be a junction, in which case the value may be of any of the conjuncted types. This represents partial type information, but not an assumed interface.

12.3 Static Type Checking

Describe what this means. Conversions that are allowed or not, under what conditions?

12.3.1 Conceptual Overview

Informative

> A variable's type is a constraint indicating what sorts of values the variable may contain. More precisely, it's a promise that the object or objects contained in the variable are capable of responding to the methods of the indicated "role". —*Synopses 2*

A variable's type is a promise.

The implementation may rely on that to optimize code.

If it relies on it, the implementation cannot tolerate a breach in that promise.

Those are the postulates. Every detail can be logically derived from them, or introduces explicit choices in the design of the language.

12.3.1.1 Effects of Typing

The effects can be divided into two categories.

Knowledge of types can allow the implementation to generate better code, and perhaps give warnings, because it knows what it can rely on. Use of this knowledge must not change the meaning; just make it more efficient.

Use of types can change the meaning of a statement. In the extreme example, it becomes an error because the types don't match.

A full specification must be clear as to when types affect semantics, in order to guarantee standard behavior across implementations. We can't have optimized programs behave differently from non-optimized programs.

12.3.1.2 Errors in type

Any attempt to store a value into an lvalue with an incompatible type is an error. Sometimes it can figure that out at compile time and reject the program:

```
sub goFishing (-->Array of Fish) { ... }
my Dog $spot;
$spot = goFishing;  #compile-time error, I hope!
```

It would be nice to know when I can guarantee a compile-time error, rather than waiting for that line to be encountered at run-time, if ever. But that is a different issue from demanding that it be an error in the first place.

12.3.1.3 Type Conversions

An assignment of one type to an lvalue expecting a different type might, rather than being an error, cause the value to be changed. This is a semantic change in the program caused by type information.

```
sub foo () is of B { ... }
my A $x = foo;
my Int $y = 2.779 * $k1;
```

The assignment to $x might be an error like in the above example. But, perhaps the relationship between A and B is such that it can be converted?

There is the case where B "isa" A. This is always allowed, and provides polymorphism.

Maybe the implementation can code

```
$x = A.new (foo)
```

or ("Any type name in rvalue context is parsed as a list operator indicating a typecast" —*Synopsis 2*)[10]

```
$x = A foo();
```

or some method of B that states it can convert the object to an A. The syntax for that is not known, but that's unimportant for the present discussion.

In the assignment to $y, the floating point value needs to be converted to an Int. Perhaps the multiplication is close to an integer anyway. You can certainly see how a conversion could be done, but it looses information.

10 But how does that list operator know what to do?

Should either of these be performed without error, or should the user explicitly state his desire for a conversion?

This is where we need to make a decision. I think the Perl way is to allow either way. In general, where there is a possibility for an automatic conversion, it might be silent, might be a warning, or might be an error. In general, you could specify this decision on a case-by-case basis.

The definition of class B might include the notation that implicit conversions are allowed to A, so that will be the case if the user does not override it in some lexical scope. The complex of built-in types concerning information loss can be done with a single command that sets the entire group, much in the say that imports can have groups.

The default is, "if I bothered to write the type information, you can tell me what you think of it." and issue a warning for wrong-way conversions. That can be strengthened to an error or turned off with one command, or have the details tuned by allowing or disallowing specific conversions among the built-in types.

12.3.1.4 Static and Dynamic type

Static type information is what is present (or inferred) at compile time. Dynamic type is the actual type of the object at run-time. The dynamic type can be more specialized because it is a derived class using polymorphism, or because the static type was constrained but not completely specified. As a degenerate case, a static type of All means no knowledge at all.

Method dispatch and multi calling is performed using the actual dynamic type of the parameters.

If type checking could not be fully done at compile time, checking will need to be done at run time using the actual (dynamic) type once it is known.

Thanks to junctions and overlapping types, it may be the case that only stores or only fetches need to be checked at run time.

We need to decide if "checking" involves outright rejection only or can convert. If the rule is different than compile-time checking, then it needs to be unambiguous when checking is guaranteed to be performed in which episode! If optimization can be done to different degrees, then it would need to track when something was supposed to be done as well as when it was really done, to give the correct behavior. So, I like the semantics to always be the same regardless of how soon it can discover the need.

12.3.1.5 Generic Types

Generic types are distinguished by being "compile time" type information that is not actually known at compile time.

Consider:

```
sub foo ();  # return type unknown
sub bar (Num ::T $a, T $b --> T)
{
T $x = foo;
}
```

The type of T is not known until the function is called. It refers to the dynamic type, so it might not be until the actual run-time of the call, not just compile-time inspection of the calling code.

That means that all the checking of whether the result of foo can be assigned to $x must be done after T is known, as well as after the result of foo is known.

Generic types are explained in detail in §12.5 on page 72.

12.3.1.6 Unspecified Types

An unspecified type is really Any.

A partly specified type, such as a role, might cause a similar situation. In general, the right-hand-side may be known, at compile time, to be of an overlapping type to the left-hand-side. This means making the final decision at run time.

Should this look at the dynamic type and apply the same type rules that it used for static types, or are they different? Or is it up to the individual RHS type to decide whether to muti-dispatch on the actual dynamic type and re-check the rules? Any is just a special case. Should it be special, or just like any other overlapping type?

12.4 Fully-Specified Types

Normally, the use of static type declarations means that a variable or subroutine return value will be of some type that is known at compile time.

```
my Fish $f;
my Animal $animal;
```

The variable $f is known to be of type Fish, and this provides the promises that are valuable for static type checking and optimization based on known type information at compile time. The compiler knows that $f has a particular interface.

Likewise with $animal. Even though type Animal may be a role that is supported by many concrete types including Fish, this is "fully specified" in that the compiler knows that $animal will have (at least) a particular interface.

However, Perl 6 allows for more interesting possibilities, not seen in other strongly typed languages. Consider the use of a Junction:

```
Fish|Bird $dinner;
```

The variable $dinner might have one type or might have another type. That information is useful for static type checking and optimization, but you cannot guarantee that $dinner has a specific interface. At run time, a particular instance

may have one interface or another interface. From a certain point of view, that is not strong typing.

When the type declaration allows for either/or capabilities, so that you can not say that the variable is always of a specific type, then this situation is called an *incompletely specified type*, and is not a *fully specified* type.

Now the compiler may figure our that Fish and Bird have several interfaces in common (Animal and Object), it may be able to optimize based on this. But the implementation will not introduce observable behavior changes based on this inference.

For example, the default initializer of $dinner, being an incompletely specified type, is the undef value. The implementation will not take it upon itself to generate a protoobject that may autovivify to a Fish or a Bird on its whim.

On the other hand, when types have a logical AND relationship, this does provide a strong promise of an interface that is always present in that variable. That is the interface that is the union of those ANDed together. If some of the types being ANDed are incompletely specified, it does not alter the promises made by the others.

```
Animal Fish|Bird $dinner;
```

In this example, the compiler will know that the value in $dinner will always have the Animal role, and thus is a completely specified type. Even though the specification is redundant, in that the Animal role is present in Fish and Bird already, being explicit about exactly what common interface you intend will allow the implementation to manifest observed behavior based on this. It considers this a completely-specified type that is further constrained (same as with subtypes) and happens to be a role rather than a concrete type.

In this case, the default initializer will be an undefined value of type Animal, which does not autovivify, and allows the compiler to assume that the Animal interface is always present, and the Failure type is not automatically ORed in.

12.5 Generic Types

A generic type is one that is not known at compile time. Perl 6 offers features for parameterized types and generic types in function signatures. But in general, you can use a type as a type even if it cannot be identified at compile time.

```
sub GetType (-->Type) { ... }

my ::RunTimeType := GetType;
my RunTimeType $x;
$x= foo;
```

Just because RunTimeType is not known at the time $x= foo is compiled does not obviate the requirement that the assignment be tested to make sure that $x fulfills it promise of only holding items that support the proper interface.

This is essentially the situation for methods of a class that has type parameters, and

any functions that have generic type parameters.

12.5.1 Class names used in methods

The name of any class, when used in a method, is generic. This is called a *virtual class name*. The meaning may be overridden in derived types by defining another class with that name.

```
module M {

  class C { ... }

  class D {
    has C $.a;
    method foo ()
     {
     my C $b.=new(5);
     say $b.HOW.name(:qualified);
     }
    } # end of D

  } # end of M

package P {

  class C { ... } # a totally different class C than the global one

  class E is D { }

  my D $x=.new;
  $x.foo;
  my E $y=.new;
  $y.foo;
  say &D::foo =:= &E::foo;  # prints True.

  } # end of P
```

In this example, the call to $x.foo prints M::C, indicating that the use of C in foo refers to the class C from the same module as D. However, $y.foo, even though the method is inherited and runs the same code as before, prints P::C indicating that now foo is using a different meaning for its use of C.

The meaning of C, in all the methods that make up a class including inherited methods from base classes, is that of the C in the lexical scope when the class is defined.

Effectively, every unqualified type name[11] used in a method that refers to a non-global type causes the creation of a virtual function that returns that Type, and derived classes override the functions if the meaning changed due to differences in scope. And, uses of the type symbol by methods are not bound at compile time but query the virtual function when the Type is needed.

The effective code for the above example is something like:

```
class D {
  has C $.a;
  my method Class ®getC () { return C }
  method foo ()
   {
   my ::®genericC := ®getC;
   my ®genericC $b.=new(5);
   say $b.HOW.name(:qualified);
   }
  } # end of D
◦ ◦ ◦
class E is D {
  my method Class ®getC () { return C } # refers to C visible here, P::C
  }
```

The virtual class name affects parameters and return types in methods too, and that would be more difficult to write equivalent code for. It is impossible to write *exactly* equivalent code[12] for that using other language features, without resorting to trickery if even then.

The symbol CLASS is also a virtual class name. The use of CLASS will always refer to the actual dynamic type of the complete object, not necessarily the class where the symbol was used.

The virtual type CLASS is automatically overridden in every class equivilent to emitting the statement at the beginning of the class:

```
my ::CLASS ::= ::?CLASS;
```

The scope of the overridden CLASS begins immediately after the class keyword, so may be used before the opening brace of the class's block. It may also be used if the class is anonymous.

Virtual class name references apply uniformly to all uses of the symbol within the scope of the class package.

A virtual class name may be used as a listop to indicate a conversion. The conversion code and the type it produces are virtual just like the type.

If the same class name C refers to different types in different places within the

11 All types are classes, regardless of the keyword used to define it.

12 In particular, repeating the methods with different types rather than inheriting them would change things: wrapping it in-place would not affect the derived class.

definition of D, the lexical regions where the use of C means different things cause different virtual class symbol to be generated. A derived class E may override some of them based on a re-analysis of the scope at the point where E is defined and any class definitions nested in E.

The partitioning of the class D into lexical regions is done when D is created. Each different original meaning of C creates a different virtual class symbol symbol.

The re-analysis of scopes and the effective generation of overrides for virtual class name lookups is done at a point *p1* just before the first method definition in E, or before the end of the class body if there isn't one.

Re-defining a class name at a point *p2* after *p1* has no effect on the overrides, and by default generates a compile-time warning. Its presence may affect further-derived classes, and of course qualified names and non-method uses within D after *p2*.

In general, for any virtual class symbol C used in class D originally bound to class C_1, and class E that inherits from D, if there exists in the scope of the declaration of E another class C_2 that transitively hides C_1 , then C_2 overrides the virtual generic C in the definition of E.

"Transitively hides" means hides, or hides something that hides, with any number of intermediate hidings.

```
module M {
  class C { ... }
  class D {
    # lexical region where C refers to M::C
    method m1 () { my C $x; my Int $y; ... }
    class C { ... }
    # lexical region where C refers to D::C
    method m2 () { my C $x; ... }
    }
  }
package P {
  class C { ... }
  class Int { ... }
  class E1 is D {
    # re-analysis done here, as if the class was written here.
    method foo () { ... }
    class C { ... }  # no effect on rebinding, but issues a warning
    }
  }
```

In the above example, the existence of P::C when E1 is defined will override uses of M::C but not D::C.

In particular, if a class C is named in a scope *s1* enclosing the definition of D, virtual

class references that originally referred to C in *s1* may be overridden by a class named C in a scope *s2* that encloses the definition of E, if *s2* transitively hides *s1* at the point where E is defined.

Meanwhile, the definition of P::Int has no effect on m1's use of the standard Int type, because types that originally resolve to global names (here *::Int) do not generate virtual class names. (If you want a global type to be virtual, make an alias in a different scope first.)

Continuing the previous example,

```
package P {
    class E2 is D {
    class C { ... }
    method foo () { ... }
    }
  }
```

Class E2::C will override the use of D::C in m2, but not the use of M::C in m1.

12.5.2 Error Checking and specialization

For types known at compile-time, much type checking is done at compile time, and the compiler can find an error before the code begins running.

For generic types, the implementation might specialize the code once it knows what the type is. This is especially the case for functions with generic type parameters and methods of parameterized classes. For example,

```
sub Foo (::T $x, T $y)
{
f1($x);
f1($y);
$x.m1;
$y.m1;
f2($x);
f2($y)
}
```

When the function Foo is called with ::T becoming class C, for example, the compiler may go through the entire function and figure out which are called, in the same manner as it does for types known at compile time. This is more efficient than doing the run-time lookup of f1 every time it is called with the same type.

> For writing such tests explicitly, we have T.can<m1>. For non-multis you can try matching the signature of a Routine object. But there is no simple test for multis. We need a way to check if there are any candidates, even if it defers actual decision until call time.

This means that as soon as ::T is bound to C, the implementation might realize that C does not have an m1 method and give an error. This means that f1 is never called.

If the implementation does not specialize the function but checks each binding at run time when it is encountered, then f1 will be called twice before the error is found.

The implementation is allowed to analyze the entire scope of a generic type at the time it is bound, and throw exceptions relating to type errors anywhere in the block, before executing the block. For generic type parameters to any block or Routine, it behaves exactly as if type checks are placed in a PRE block inside that block. For type variables bound inside a block, it behaves as if assertions are coded immediately following the := expression of the entire signature.

> Need an assertion that fails in the same way as a PRE or POST block, but called whenever you want.

It is not allowed to execute some lines, then give an error for something even farther down without executing all the code up to that point. The pre-run error check is done only before executing the code in the entire block.

However, this bind-time checking does not have to find all errors, and is not required of the implementation. Some static analysis might be done, and other checks performed at run-time.

As a special case, the programmer can seemingly disable bind-time type checking using a pragma (*which?*). This is useful to fix a program that ran on one implementation but failed on another that had more optimization and analysis, of if you know ahead of time that you do want to perform some preparatory work before diagnosing the error.

When this pragma is in effect, an implementation can still analyze things and generate optimal code. But in the expression where the error is known to exist, generate the run-time checking code instead. This will behave as if the entire block did run-time checking, and will allow the code to work if something funny is going on that the static type analysis missed.

12.5.3 Code object identity and specialization

A function that has a generic type parameter is still a single Routine.

```
sub f1 (Num ::T $x, T $y --> T) { ... }
sub f2 (::SomeType, Int $n --> SomeType @) { ... }
```

Function f1 takes two arguments and grasps the actual type of the first argument implicitly as a generic type parameter. Function f2 takes two arguments, one of which is explicitly a type. Either way, both functions declare a generic type as a formal parameter that is in scope for the entire function's code block, and the implementation may find it useful to specialize the code, generating a different compiled object for each value of the generic type, as opposed to checking everything at run time in a universal implementation.

However, this optimization is something that must remain internal to the

implementation. Semantically, &f1 is a single Routine object, regardless of it having generic type parameters.

```
my $func = &f1;
my $result = $func(2,4);
say $result.WHAT;  # outputs "Int"
$result = $func("hello","world");
say $result.WHAT;  # outputs "Str"
```

On the other hand, the use of CLASS in a role is generic, because it refers to the actual class that eventually composes that role. But, methods in different classes will be different Routine objects, even if they are both composed from the same Role. *Realizing* a method from a role generates a unique Routine object. Since they are different Routines anyway, the implementation can specialize it for different values of CLASS within the different Routines.

```
role R1 {
  method f3 (::?CLASS $self, $x) { ... }
  }
class C1 does R1 { }
class C2 does R1 { }

Routine $r1 = C1.can("f3")[0];
Routine $r2 = C1.can("f3")[0];
say $r1 === $r2;  # outputs "False"
```

Also, wrapping C1::f3 in place does not affect C2::f3, or vice versa.

12.6 The Failure type

The special Failure object appears to violate the strong type checking.

A value having the Failure role may be assigned to variables of any non-compact type.

```
my Int $x;
$x = fail ("does not compute");
```

However, this is still done with static type knowledge. A Failure value may be implicitly converted to any non-compact type. The result of the conversion is the default undefined prototype of the proper type that has the Failure trait applied to it. So the above is equivalent to:

```
my Int $x;
my Failure $temp = fail ("does not compute");
$x = Int but Failure($temp);
```

This requires that the type of $x be fully-specified, since it needs to know what kind of undefined object to create. So for incompletely-specified types, the Failure type is always automatically ORed in.

```
my Int|Str $j;
```

```
$j =  fail ("does not compute");
```

Is really compiled as:

```
my Int|Str|Failure $j;
$j =  fail ("does not compute");
```

The special behavior of Failure does not require that every use of a strongly-typed variable be checked for the proper interface. Rather, it knows at compile-time that a Failure value or non-typed value is being assigned to a strongly-typed variable, and only has to do such checking when Failure is a possibility.

Once it is converted, the value is indeed of the proper type, so no further testing needs to be done.

```
my $x;  # untyped
° ° °

my Int $y = $x;  # type checking needed
my Int $z = fail("does not compute");  # knows to convert Failure to Int.
$y = $z;  # failure object assigned, but is already an Int, so type is correct.
```

12.7 Type equivalence

Types are officially compared using name equivalence rather than structural equivalence. However, we're rather liberal in what we consider a name. For example, the name includes the version and authority associated with the module defining the type (even if the type itself is "anonymous"). Beyond that, when you instantiate a parametric type, the arguments are considered part of the "long name" of the resulting type, so one `Array of Int` is equivalent to another `Array of Int`. (Another way to look at it is that the type instantiation "factory" is memoized.) Typename aliases are considered equivalent to the original type.

This name equivalence of parametric types extends only to parameters that can be considered immutable (or that at least can have an immutable snapshot taken of them). Two distinct classes are never considered equivalent even if they have the same attributes because classes are not considered immutable.

12.8 Type checking

Type A is **consistent** with type B in the following cases:

1. If A is the same type as B.
2. A is the special Any type.
3. If A and B are both classes and A “isa” B through inheritance and such inheritance is not hidden or otherwise explicitly disabled as an “isa” relationship.

4. If A is a role and B is a class supports that role.
5. If A is a role and B is a role that includes[13] A.
6. If B is an OR junction, and A is consistent with every type in that junction.
7. If A is an OR junction, and every type in A is consistent with B.
8. If B is a parameterized class or role:

Takes quite a bit of explaining...

Type A and type B **overlap** in the following cases: (A and B may be in either order. Symmetrical cases are omitted for brevity)

1. If A is an OR junction and B is a type that is consistent with at least one of the types in A.
2. If A is an OR junction and B is an OR junction, and at least one of the types in B overlaps with A.
3. A and B have at least one role in common.
4. B is a subtype, and A overlaps B's underlying type.
5. A is a subtype, and A's underlying type is consistent with B or B is consistent with A's underlying type.

12.9 Container type

The container type must support the proper role based on the sigil it is bound to.

Sigil	**role**	**default container**
$	Tie::Scalar	Scalar
@	Tie::Array	Array
%	Tie::Hash	Hash

(standard role names and contents are *proposed*)

It is erroneous for the container implementation to return a value whose actual type is inconsistent with the value type of the variable it is accessed through.

When a container is constructed, it is provided information about the declared type of the variable it is initially bound to. The implementation of the container is must throw an Exception::WrongType exception (standard exception type *proposed*) if it cannot support the declared type.

The various Tie:: roles include a function that is called at compile time with the declared value type. This returns a Failure object, which will be used at compile compile time, if the container class determines that the declaration is in error. (*proposed*)

13 Need to decide on the correct formal term here.

12.10 Variable Binding

If a variable is bound to an existing container without using the :defer adverb, the variable's declared value type must be consistent with the actual value type of the container. In the absence of the :coerse adverb, the variable's declared type must be consistent with the static type of the container, and this must be checked at compile time.

If the binding is performed with the :defer adverb, then the variable's declared value type must overlap the actual value type of the container. In the absence of the :coerse adverb, the existence of an overlap is checked at compile time with the static type of the container, and fetch and store run-time tests may be formulated at compile time, and must use only the static type of the container[14]. With the :coerse adverb, fetch and store tests must consider overlap between the variable's declared type and the actual value type of the container.

```
class C { ... }
class D is C { ... }
my D $d1;  # creates container with actual value type of D
my C $c1 := $d1;  # OK, since D "isa" C.
my D $d2 := $c1;  # compile-time error, since static types are compared
my D $d3 := $c1 :coerce;  # OK at compile time because test explicitly deferred to run time.

my Int|Str $x1;
my Int $x2 := $x1;  # error, since could get Str's out depending on actual value
my Int $y1;
my Int|Str $y2 := $y1;  # error, since could put Str's into container of Ints
my Int $x3 := $x1 :defer; # OK, fetches checked at run-time
my Int|Str $y3 := y1 :defer;  # OK, stores checked at run-time
my Int $x4 := $x1 :coerce;  # fails at run time because actual container type still wrong
my Int $x5 := $y2 :coerce;  # type check agrees at run time.
```

(The :coerce and :defer adverbs to := is *proposed*)

If binding type checking is deferred until run time, the actual value type of the container is checked when the binding takes place. If the explicit deferral is redundant because the static types match anyway, the run-time test may be optimized out, and a warning may be issued at compile time.

If the implementation can tell at compile time that the run-time check will always fail, then it is allowed to emit an error at compile time. It is also allowed to emit a warning at compile time.

If the type checking is performed at run-time, it throws an Exception::WrongType exception. (standard exception type *proposed*)

A binding may be performed when the container's value type overlaps with the variable's declared value type, if explicitly marked.

14 That is, the program will give the same results regardless of whether decisions about the testing were made at compile time or performed only at run time.

The implementation must check every store into a container against the declared value type of the variable through which it is accessed, and against the actual value type of the container. When this can be done at compile time, run-time checking can be omitted. A combination of static type analysis and run time checking may be required if the container's value type overlaps the variable's declared value type.

The implementation is allowed to assume that any value retrieved from a variable has an actual dynamic type that agrees with the declared value type. That is, the declared value type of the variable is used as the static type of any value retrieved from it, for subsequent static analysis of the types in the expression it appears it.

If the value type of the container differs from the declared type of the variable through which it is accessed, and deferred testing was explicitly requested, then the process of retrieval must include type checking. If the actual value is not consistent with the declared value type of the variable (when :defer is used) or with the overlap between the declared value type and the static container type (when :defer is not used), it throws an Exception::WrongType exception. (standard exception type *proposed*)

Note that if the ::= binding syntax is used, then "when the binding takes place" is earlier and the "run time" of performing the binding happens to be before the next statement is compiled, and the implementation may be more sure about knowing the actual container type at compile time, but the rules are no different. It just offers more room for optimization.

12.11 Parameter Binding

Parameter binding is looser than variable binding with := because it allows pass by copy/return. "Ref" binding is the same as :=.

Need to describe when it may or may not make copies, in light of possible side effects and possible extreme overhead.

13 Signatures and Captures

13.1 What the Synopses Says

First step: catalog what is known about them.

From S02:

A signature object (`Signature`) may be created with colon-prefixed parens:

```
my ::MySig ::= :(Int, Num, Complex, Status)
```

Expressions inside the signature are parsed as parameter declarations rather than ordinary expressions. See S06 for more details on the syntax for parameters.

Signature objects bound to type variables (as in the example above) may be used within other signatures to apply additional type constraints. When applied to a `Capture` argument, the signature allows you to take the types of the capture's arguments from `MySig`, but declare the (untyped) variable names yourself via an additional signature in parentheses:

```
sub foo (Num Dog|Cat $numdog, MySig $a ($i,$j,$k,$mousestatus)) {...}
foo($mynumdog, \(1, 2.7182818, 1.0i, statmouse());
```

With multiple dispatch, `&foo` may not be sufficient to uniquely name a specific function. In that case, the type may be refined by using a signature literal as a postfix operator:

```
&foo:(Int,Num)
```

It still just returns a `Code` object.

The `|` prefix operator may be used to force "capture" context on its argument and *also* defeat any scalar argument checking imposed by subroutine signature declarations. Any resulting list arguments are then evaluated lazily.

Signatures on non-multi subs can be checked at compile time, whereas multi sub and method call signatures can only be checked at run time (in the absence of special instructions to the optimizer).

This is not a problem for arguments that are arrays or hashes, since they don't have to care about their context, but just return themselves in any event, which may or

may not be lazily flattened.

However, function calls in the argument list can't know their eventual context because the method hasn't been dispatched yet, so we don't know which signature to check against. As in Perl 5, list context is assumed unless you explicitly qualify the argument with an item context operator.

From S03:

Smart matching:

```
    $_         X         Type of Match Implied   Match if (given $_)
    ======     =====     =====================   ===================
    Signature Signature sig compatibility       $_ is a subset of X      ???
   Code      Signature sig compatibility       $_.sig is a subset of X  ???
    Capture   Signature parameters bindable     $_ could bind to X (doesn't!)
    Any       Signature parameters bindable     |$_ could bind to X (doesn't!)
    Signature Capture   parameters bindable     X could bind to $_
```

Matching against a `Signature` does not actually bind any variables, but only tests to see if the signature *could* bind. To really bind to a signature, use the `*` pattern to delegate binding to the `when` statement's block instead. Matching against `*` is special in that it takes its truth from whether the subsequent block is bound against the topic, so you can do ordered signature matching:

```
    given $capture {
        when * -> Int $a, Str $b { ... }
        when * -> Str $a, Int $b { ... }
        when * -> $a, $b         { ... }
        when *                   { ... }
    }
```

This can be useful when the unordered semantics of multiple dispatch are insufficient for defining the "pecking order" of code. Note that you can bind to either a bare block or a pointy block. Binding to a bare block conveniently leaves the topic in `$_`, so the final form above is equivalent to a `default`. (Placeholder parameters may also be used in the bare block form, though of course their types cannot be specified that way.)

Strings, arrays, lists, sequences, captures, and tree nodes can all be pattern matched by regexes or by signatures more or less interchangably.

Variable declarators such as `my` now take a *signature* as their argument.

... details on declaring variables ...

The `->` "pointy block" token also introduces a signature, but in this case you must omit both the colon and the parens.

If a signature is assigned to (whether declared or colon form), the signature is converted to a list of lvalue variables and the ordinary rules of assignment apply, except that the evaluation of the right side and the assignment happens at time determined by the declarator.

If a signature is bound to an argument list, then the binding of the arguments proceeds as if the signature were the formal parameters for a function, except that, unlike in a function call, the parameters are bound `rw` by default rather than `readonly`.

S05:

Basically, place-holder variables make an implicit signature.

The `return` function preserves its argument list as a `Capture` object, and responds to the left-hand `Signature` in a binding. This allows named return values if the caller expects one:

The want function shows some methods on the signature object.

S12:

The "long name" of a subroutine or method includes the type signature of its invocant arguments. The "short name" doesn't.

You can use an anonymous subtype in a signature:

```
sub check_even (Num where { $^n % 2 == 0 } $even) {...}
```

13.2 Variable Declarations

14 Packages and Modules

Summarize all the things that "are" packages, what they have in common, and what other kinds add to it.

Include coverage of "module", but summary only of those kinds that have their own high-level sections.

15 Classes

𝔏-items are executed immediately. That is, ordinary statements within the block forming the class body.

𝔐-items (method-like) execute instructions for the metaclass object to build the Class.

□-items (sub-like) define symbols in the scope of the block forming the class body.

Undeveloped idea: "class boxes". Goes with extensible classes. See "*Classboxes: A Minimal Module Model Supporting Local Rebinding*" by Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts

15.1 Items

15.1.1 𝔏-items

𝔏-items are executed immediately. That is, ordinary statements within the block forming the class body.

15.1.2 □-items

□-items (sub-like) define symbols in the scope of the block forming the class body.

my class — lexically scoped class.

our class — nested class (after all, classes are packages).

(likewise with any construct that creates a type, including subset, enum, …)

my constant $random = rand; — lexical scope.

our constant $random = rand; — package scope.

my $var — lexically scoped variable

our $attribute — class attributes.

{ proto, multi, only } sub

15.1.3 𝔐-items

𝔐-items (method-like) execute instructions for the metaclass object to build the

Class.

15.1.3.1 Declarators

These are things that define a name using my, our, or has. But sometimes the our keyword can be omitted.

my method — private method.

my ^method — private class method.

our method — normal method.

our ^method — normal class method.

our submethod — submethod.

my submethod - ??

has $.attribute — attribute w/accessors.

has $!attribute — attribute without generated accessors.

has $attribute — lexical alias for $!attribute.

my $.var — class attribute, plus generates accessor, not inheritable. (The variable itself or the accessor is not inheritable?)

my $!var — class attribute, no accessor, not inheritable. How is this different from a normal lexical variable? Role composition can show differences?

our $.var — class attribute, accessor, inheritable.

our $!var — class attribute, no accessor, inheritable.

15.1.3.2 class traits

class Dog is Mammal — base class.

class Dog does Pet — Role composition.

class Dog is rw — other instructions to the metaclass.

is { rw, trusts, open, nonfinal }

hides Base

15.2 Special Members

15.2.1

16 Roles

Create components that are designed for reuse, rather than for instantiation.

16.1 The CUB paper

Informative

What Perl 6 calls Roles are what the authors of "*Traits: Composable Units of Behaviour*"[15] termed traits. In Perl 6, the term "trait" means something different. In the following discussion, this paper is referred to as CUB, and the Perl 6 nomenclature from this specification is used even when referring to concepts from CUB[16], unless otherwise indicated.

In CUB, Roles are ignored after composition and are not used in determining the relationship between classes. In Perl 6, Roles can be used much like "interfaces" are in Java, indicating common features among classes.

In CUB, Roles do not specify any state nor refer to state directly. In Perl 6, Roles may include instance data (*attributes*).

In CUB, Roles have the "flattening property", meaning that a class formed by combining roles behaves identically as a class in which all the composed methods were defined directly in the class. In Perl 6, you get the same effect if you realize that the Role body adds another layer of scope. If you create a class by copying the methods from the Roles, *including a set of braces around all the stuff pulled in from a single Role*, you will get the same behavior. That is, method bodies might be closures over locally scoped symbols defined in the Role's lexical scope. And of course this does not apply to features that explicitly address Roles, such as use of the ::?ROLE symbol.

The CUB advocates aliasing over qualified Role references. One reason in particular is applicable to Perl 6: "Named trait references require the trait structure to be hard-coded in all the methods that use them. This means that changing the trait structure, or simply moving methods from one trait to another, potentially invalidates many methods." You can get the effect of aliasing, and avoid hard-coding qualified names, in the usual manner of Perl 6 aliasing:

```
my ::AliasName := Role::Qualified::Name;
```

and use AliasName within the class. If you move the implementation to another Role, only this one line needs to be changed. Note however that if you are indeed heavily

15 http://www.iam.unibe.ch/~scg/Archive/Papers/Scha03aTraits.pdf
16 That is, I'll call them Roles, to avoid confusing differing meanings of "traits".

using something from a Role, it is probably composited directly into your class anyway! Qualified references are for glue code that adapts conflicts or extends a method that comes in via a Role.

16.2 Parsing and storing a Role

The body of a Role contains various items, just like a class (§15.1, p.87). An ℒ-item in a Role is an error.

The body of a Role is parsed according to the current grammar in scope at the time the Role is defined. That is, it knows what custom operators exist, macros, and syntactic warpage. But, after creating the AST, many of the symbols are left unbound, and static type analysis is not performed. This is deferred until the items in the rule are *realized*.

Types are treated in the same manner as with classes, either binding or producing virtual class names. However, the original meaning of a virtual class name is allowed to be undefined, if the name was not found during lookup.

Symbols with qualified names are bound.

Symbols that would resolve to global scope are bound.

Symbols that would resolve to items declared using my anywhere within the lexical scope of the Role are bound.

Otherwise, unqualified names without twigils that do not refer to types are not bound, and all others are bound normally as with a class. If it is ambiguous whether a bare name in an expression refers to a type or a function[17], it is treated as a function. Use a :: sigil to indicate use of a type in this context. E.g.

```
callme(name);
```

The names callme and name are both assumed to be functions and will be looked up when the role is realized. The existence of symbols by this name outside the scope of the role will not influence this decision. If there happens to be a class name known at the point where the role is defined, that might give a warning (maybe the programmer intended callme(::name) ?), but the role is parsed to assume that name will be a function to be found when the role is composited into a class.

The implementation is allowed to generate errors at this point if it is certain that no usage of this Role can be correct, analyzing those types that are resolved immediately. But, unlike a class, the code does not have to be correct with the original meaning of virtual type names.

```
module M1;
class C {
  method makeit (Int $ --> C)  { ... }
  }
```

17 I'm not certain yet that bare words will always be taken as functions or conversion operators in an expression, as opposed to where types are expected is always in a declaration.

```
role R1 {
  has M1::C $.a .= makeit("hello");
  has M1::X $.b;
  method foo ()
  {
   my C $x .= makeit("hello");
   func_bar;
  }
  method bar (X $param --> X)
  {
   my X $temp = $param;
  }
 }
```

The use of makeit in the initializer of attribute $.a may trigger an error in the role when the role is defined, just as the same issue must trigger an error in a class. The meaning of C is definitely known to be M1::C, and the parameter is the wrong type. If the implementation chooses not to note this error when R1 is defined, it will note it when R1 is composed into a class, during the realization step.

The construct inside foo that uses C as an unqualified name is not an error[18] inside a role, although it would be an error if it appeared inside a class[19]. Unlike classes, roles are not assumed to be valid alone and in the context in which they are declared. R1 only works if composed into a class that overrides the virtual class C with something suitable, or if the class C has changed to add this method since R1 was compiled.

The definition of attribute $.b is an error when the role is being defined, because M1::X is not resolved. Because it is qualified, X is not a virtual type name. But the uses of X in bar do not cause errors, although this would have been an error in a class. X here is a virtual class name, and in roles they are allowed to be undefined originally. Again, it only works if R1 is used where X gets overridden, and this will be found at realization time.

The call to func_bar, a function which does not exist in scope when R1 is defined, is not an error. It doesn't matter whether func_bar exists or not, since it is left unbound. The name is not looked up until the role is realized. This is a different concept than overriding virtual class names. Naturally, if func_bar is not found when the code is realized, it fails at that point.

16.3 Compositing

16.3.1 Semantics of compositing

The implementation only considers definitions of things as they are known at the

18 It is OK to issue a warning.
19 Conjecture: maybe a class or individual method can be defined with is abstract.

time of compositing.

16.3.1.1 Removing duplicate Roles

The sibling Roles in a composition are independent of position. Duplicating or rearranging them will have no effect on the meaning. But hierarchical Roles do have some meaning in that Roles can override or resolve conflicts among subroles.

Consider a list of sibling Roles *RL* nominated for compositing in *T*, which may be a class or another role.

If *RL* contains more than one mention of the same (w.r.t. =:=) Role, duplicates are removed. That is, the *RL* is set-like: unordered and incapable of duplication.

For each Role *R* directly in *RL*: if there exists a Role *M* directly in *RL* such that *R* is a transitive subrole of *M*, then *R* is removed from *RL*. A transitive subrole is a subrole, or a subrole of a subrole, to any number of intermediate levels.

16.3.1.2 Building the composite

After removing duplicates, the various items defined in the Roles in *RL* are copied into *T* . Note that if a Role R has subroles, they have already been composited into R, and in the remainder of this section "items in R" refer to the composite.

The items of interest in the Roles in RL are described below, and will be referred to as ℝ-items (for items in Roles). ℝ-items include all 𝕄-items and those □-items that are declared with the our declarator. □-items that have no explicit declarator but are implicitly our are included. □-items that are declared with the my declarator are not included.

For each ℝ-item *I* directly defined in the composite, every item in the Roles in *RL* that are hidden by *I* are *marked* for non-inclusion in the composite.

For each ℝ-item *I* defined in some Role in *RL* not already marked, consider the set of unmarked ℝ-items in all the Roles in *RL* that would conflict with *I*, plus *I* itself. Items conflict if they cannot be specified in the same scope.[20] Perform the following steps on each such set:

> Mark all items in this set as already processed.
>
> > Right now it is assumed that multi's are non-conflicting if the Long Name is different. Revisit this if the situation is more complex.
>
> A set does not contain duplicates. If the same item was copied via use of the same subrole, and now appears in more than one Role in *RL*, they are still the *same* item (accessible via multiple qualified names) and is present in this set only once.
>
> If not all items in the set are auto-generated accessors, remove the auto-

20 Note that the existence of a proto declaration in the target scope can affect whether a group of items conflict.

generated accessors. Remove all items having a priority trait of -Inf.

If the set contains only *I*, copy it to the composite.

Otherwise, see if one item in the set possesses a priority trait that is strictly greater than all other items on the list. If so, copy that item to the composite.

If no unique item in the set was identified, the compositing fails. (*Which error?*)

16.3.1.3 Reference to not-copied items

An item *I* that was copied into *T* may contain references to items that were not copied.

If *I* refers to any my-items that were declared with my, then the code in *I* was "closed over" reference to those symbols. The my-items names are not made available to any additional scope.

The code directly in *T* (a class or another role) may refer to items in *RL* by using qualified names. If reference is made to an item that was not copied into *T*, then that item is copied into *T* but the name is not added directly to *T*. It will only be available using a qualified name. Items so-copied remain available using the qualified name at run time. Items that were never copied are not available at run time.

Note that items copied into *T* as qualified-name only are still copied and processed in the normal way. In particular, resolving symbols and references within those items is no different than in items copied during the compositing step.

16.3.1.4 Realization of the Composite

If the composite *T* is a Role, then the copied items stay in their unbound state with no further processing. Note that it does not record the qualified names to items that were copied from subroles. Qualified names used by the items directly in *T* were resolved during compositing, but are not recorded for discovery during run time. The remainder of this section applies to classes.

If *T* is a class, then the items that were copied into *T* from the Roles are then *realized*. To realize an item, the unbound symbols are bound, as if the item were defined in *T*. The resulting realized items are processed in the same manner as items defined in the class: symbols are added to package *T*, and commands are sent to the metaobject to define the realized 𝓜-items in the class.

Note that "binding" in this context does not necessarily mean that the symbol is bound directly to a container, as with the ::= operator. It could become a virtual class name, or a generic type. It means that now is when it interprets the names. "Binding" here refers to the AST operation, not the ::= operator.

If some unbound symbols could not be bound, realization fails. (*Which error?*) This indicates that required symbols were not defined in scope when the

compositing was performed. Static type analysis is performed on the realized code. The compositing fails if there are type mismatches or non-existent method names. (*Which error?*)

Nested packages are created inside *T* to reflect the qualified names of items that were copied. A run time, you can inspect the scopes to see which directly-included role an item came from, but not which subrole was used to form a direct role.

When a code item is realized, a distinct object is generated. Different realizations of the same method from a single role will compare as different Routine objects[21]. Note that this is different from the behavior of generics.

16.3.2 Resolving conflicts

Composition order is irrelevant. Therefore, conflicts can't be resolved using sibling ordering.

16.3.2.1 Priority

If conflicting items exist, one might be preferred. Automatically generated accessors will yield to defined methods. Items may have different assigned priorities.

The priority trait may be declared on any item in a Role. If no such trait is declared, the normal priority of an item is 0. If you declare the priority trait without giving a parameter, the priority is 1. This is the normal case to indicate "high priority", such as writing a Role whose purpose is to resolve conflicts among sibling roles.

```
method foo () is priority { ... }
```

If you want a role to supply a default item that bows out if others are present, in a similar manner to the way automatically generated accessors are, define it with !priority. This assigns a priority value of −1.

```
method foo() is !priority { ... }
```

If you need finer-grained control, you can give a parameter. The parameter is a Rat, so you can give any fraction around existing values, but it doesn't have the issues with equality testing that fractional Num values have.

```
method foo () is priority(3 div 5) { ... }
```

Is there a better way to write a Rat literal? 3/5 is defined as doing Num (floating point) division. Maybe it should be more type-aware and convert to Num. The operator div makes me think of integer division as in Pascal.

The priority trait on an item of a Role might or might not be visible on the realized item in a class.

21 If they end up being identical because the context is the same for symbols that need to be bound, the implementation can certainly share bits internally. But they must still be distinct objects. The behavior of .wrap, for example, must behave as specified.

16.3.2.2 Exclusion and Adjustment

Items can be excluded from a Role. This can resolve conflicting items by saying which ones you don't want.

```
does R1(:exclude<m1 $.var>)
```

This effectively resets the priority of those items to −Inf. So, they are removed from consideration, but are still available as qualified references.

Rather than remove an item from every Role except one, it can be handier to just indicate the one you do want:

```
does R1(:specify<m2>)
```

This sets the priority to +Inf. It has the further effect of giving a warning (*which?*) if you defined the item directly so it is not taken from the Role after all.

You can also specify a specific item and adjust its priority:

```
method m1 is priority(7 div 8) ::= R1::m1;
```

When defining a role, declaring items using ::= to bind to subrole items has the same effect as copying that item, and can have the added effect of aliasing it to a different name and changing the priority trait.

16.3.2.3 Redefining

A class or role can define an item, and that prevents conflicting items from being copied from roles at all. For code items, this can combine behavior from Role items by referring to their qualified names.

```
method m3 ($x)
{
 dothisbefore;
 R1::m3($x);
 &R2::m3.callsame;
 dothisafter;
}
```

But to just redirect to one, it could be more efficient to use :specify, or to alias it. This is good for non-code items, where combining doesn't make sense.

```
has SomeType $.a1;                  # define it myself, ignores $a1 in any Roles.
has SomeType $.a2 ::= R2::<$.a2>;  # explicitly pull down this one
```

For non-code items, aliasing rather than redefining might not be worth anything when declaring a class. But in declaring a role, making sure the *same* item is copied from a subrole can avoid conflicts.

17 Calls

This details the semantics of calling and returning from functions of all kinds.

17.1 sub calling lookup

This explains what happens when a function is invoked "as a sub".

This occurs in the following circumstances:

1. Function call notation: func(1,2,3). Note that &func(1,2,3) is different: it looks up a Code object named &func based on the ordinary lexical name lookup only, then applies the call to the result.
2. List-op notation: func 1,2,3.

A candidate list of subs is located using the rules as detailed in §17.8 on page 98. If there are no candidates, the call fails (*which exception?*).

The candidate list becomes the "current call candidate list" for purposes of callnext etc., temporarily replacing the existing "current call candidate list" for the duration of the call. The first candidate is then removed from the list and invoked.

17.2 ordinary method calling lookup

This explains what happens when a function is invoked "as a method call", but excludes private methods.

This occurs in the following circumstances:

1. Use the dot notation: $obj.method(1,2,3), or with the implicit $_ for the invocant, .method(1,2,3). Note that the parentheses can be omitted if there are no arguments.

   This includes methods named with an extended identifier (e.g. $obj.prefix:<!>), even when the syntactic category is omitted (e.g. $obj.:<!>). But the dotted form of postfix and postcircumfix is operator notation (see §17.3 on page 97), not method calling syntax.

   This includes method names beginning with a colon (e.g. $obj.:!x)
2. Using a method name beginning with a colon with the implicit invocant. This includes when they are "stacked" with an implicit junctive "and" (e.g. when :r :w :x)
3. Use indirect object notation: method $obj: 1,2,3. Note that the colon after the invocant is required, even if there are no arguments.

If the invocant is in item expression context, then the method is applied to the value type, not the container. Otherwise, the method is applied to the container object, and "the type" below refers to the container's type.

For example, $x.pop means to call the pop method on the value stored in $x, not on

the Scalar object bound to the $x symbol. On the other hand, @x.pop means to call the pop method on the Array bound to the @x symbol, not on any of the values stored in @x.[22]

The ordinary method calling lookup will first will attempt to locate the named method in the object based on the actual type of $obj, and the argument list, as detailed in §17.6 on page 98.

If no candidates were found, it then performs a sub lookup. The sub lookup treats the invocant as the first positional parameter, and then uses the rules as detailed in §17.8 on page 98. For example, $obj.foo(1,2,3) becomes foo($obj,1,2,3).

If there are still no candidates, the call fails (*which exception?*).

The candidate list becomes the "current call candidate list" for purposes of callnext etc., temporarily replacing the existing "current call candidate list" for the duration of the call. The first candidate is then removed from the list and invoked.

17.3 operator notation calls

An operator may be infix, prefix, postfix, circumfix, or postcircumfix.

As explained in XXXXX, an operator name may contain letters, symbols, or a mixture.

The syntax of the operator determines how its arguments are located in relation to the operator token. Once this is done, the call is normalized as a function with an extended identifier and an argument list.

A list of candidates is formed by merging the lookup for both sub lookup (§17.6 on page 98) and method lookup (§17.8 on page 98). For method lookup, the first argument is treated as the invocant, using the value type if the argument is in expression item context, and the container type otherwise, as with invocant arguments called using the ordinary method calling lookup.

The candidates are merged using a stable ordering, with methods preferred over subs if the rank is the same.

> This needs work. If the ordering of methods is based on lexical hiding, what sense is merging? Choosing the best match from the top of either list is sensible. But callnext etc. can see the whole list so it must exist. Also, argument namespace-dependant lookup is probably not needed as in C++ because "exported" operators are always global and multi. Is the a concept of a non-member defined with the class? Yes, it could be a sub not a method. This might be needed if the self-like argument is not first. Not sure if conversion symmetry is an issue ... but maybe the issue is just different (but still present).
>
> S06 says that you should never install non-standard operators into *. Names may be in any scope, but syntactical effects are lexically scoped. So, searching doesn't make sense for non-standard operators since you could not parse it unless it existed in the lexical scope anyway.

22 But the hyperoperator notation @xÂ».pop would call pop on each value stored in @x.

If there are no candidates, the call fails (*which exception?*).

The candidate list becomes the "current call candidate list" for purposes of callnext etc., temporarily replacing the existing "current call candidate list" for the duration of the call. The first candidate is then removed from the list and invoked.

17.4 private method calling lookup

17.5 indirect method call

17.6 method candidate selection

The list of candidates are methods that could handle the request. This includes inherited members and variants of multi methods. However, the candidate list does not contain all the methods in the search that have the right name. It uses parameter matching to list only those that could be called according to the analysis detailed below.

Note: submethods are always constrained to an exact match of the invocant type.

Searching is done using the declared (static) type of the invocant, unless it is Any. Junctions check "isa" against the declared possibilities more-derived first (partial ordering). For Any, the dynamic type is searched.

17.7 method call semantics

17.8 sub candidate selection

Note: includes submethods.

18 Grammars

19 Exceptions

19.1 Standard Exception Types

Exception

Exception::WrongType

20 Macros

21 POD documentation

22 Garbage Collection

Details. Finalizers, hooks, etc.

23 Special Variables

23.1 Magical lexically-scoped values

23.1.1 &?BLOCK

&?BLOCK is always an alias for the current block, and has the static type of Code. If the innermost lexical block happens to be the main block of a Routine, then &?BLOCK just returns the Block object, not the Routine object that contains it.

This example specifies recursion on an anonymous block:

```
my $anonfactorial = -> Int $n { $n < 2
        ?? 1
        !! $n * &?BLOCK($n-1)
        };
```

23.1.2 $?LINE

23.1.3 &?ROUTINE

&?ROUTINE is always an alias for the lexically innermost Routine (which may be a sub, method, or submethod), and has the static type of Routine.

Note that &?ROUTINE refers to the current single sub, even if it is declared multi.

This example specifies tail-recursion on an anonymous sub:

```
my $anonfactorial = sub (Int $n) {
        return 1 if $n<2;
        return $n * &?ROUTINE($n-1);
        };
```

24 Library

24.1 ubiquitous functions

These are reserved names that are function-like in how they are parsed. But, they have a built-in meaning that may be specific to where in the program text they are called from. They are reserved for the implementation as functions in any scope.

They may be implemented as actual macros, as functions that are defined automatically in every scope with the proper meaning for that scope, as compiler keywords with no actual library presence, or some combination of the above.

They may not be used in any manner other than to call them. You cannot reference the function symbol itself, wrap it, etc. This is to give the implementation maximum freedom in how they work, and to allow hybrid approaches where there are indeed actual functions or macros present but special implementation knowledge of their semantics as well.

You can declare methods and submethods with these same names, but not subs. When called using the method-call syntax, the names are looked up normally. If called as a sub, the special meaning is used, as if it was declared as a sub in the same scope as the point of the call.

24.1.1 caller

24.1.2 callsame

Update code to also work with method candidate lists.

This behaves like the Callable.callsame method, but automatically knows what the target Callable object is. It only works inside the call to code that was called through a Callable object that was modified by its .wrap method.

It may be used by the code that was installed by .wrap, and code called within that dynamic scope. But it is not visible to the original function that was replaced by the .wrap. Specifically, the dynamic scope starts with the code that was installed by .wrap, and ends at any call to callsame, callwith, nextsame, or nextwith.

If the code called by callsame etc. is itself another wrapper, than it has its own target Code object by virtue of that wrapping.

```
sub foo ()
{
 callsame;  # CALL 1
}

sub helper ()
{
```

```
 callsame;  # CALL 2
 }

sub replacement ()
 {
 helper;
 }

&foo.wrap(&replacement);
&foo.wrap( {callsame } );  # CALL 3
foo;
replacement;
```

When foo is called at the bottom of this listing, the outer wrapper refers to callsame, and that acts as the .callsame method on the Callable object that was replaced by the call to .wrap. So, it calls replacement. Then the line marked CALL 2 is executed, and since that is within the dynamic scope of code that was installed as a wrapper, it works and calls .callsame on the original &foo code block. Finally, the line marked CALL 1 is encountered, and that is an error because it is not a replacement—the wrapper ends and the wrapped code begins with CALL 2.

This error can only be detected at run-time because it looks just like CALL 2 at compile time. It is correct only if the code gets called as part of a wrapper.

If replacement is called explicitly, then CALL 2 fails because it is not executing as part of the context of a wrapper installed by .wrap.

Likewise, given

```
sub bar ()
 {
 say "bar is called";
 }

&bar.wrap(&replacement);
bar;
```

Then when CALL 2 is encountered, it knows that it should act on the original &bar code block. So the same line CALL 2 may be an error, locate &foo's original code block, or locate &bar's original code block, depending on how it was called.

24.1.2.1 How might it work

Informative

The implementation of the .wrap method sets the stage. It does not use the supplied closure directly, but wraps its own processing around it first:

```
# inside class Routine
method wrap (&block --> RestoreHandle)
```

```
{
my &original := .do;
my RestoreHandle $retval = .new( {self.do = &original} );
my $newcode = {
  my &®wraptarget is context := &original;
  &block.callsame;
  }
.do = $newcode;
return $retval;
}
```

That doesn't support unwrapping something in the middle, and the context variable $®wraptarget is visible[23].

And the implementation of .callsame on the same object needs to turn it off again:

```
method callsame ()
{
if defined $+®wraptarget {
  if &+®wraptarget === .do {
    &+®wraptarget = undef;
    }
  }
# proceed with actual implementation.
 ◦ ◦ ◦

}
```

Now the implementation of the non-member form of callsame can be roughly equivalent to

```
&+®wraptarget.callsame
```

24.1.2.2 Also works in method candidates

From S12: Any method can defer to the next candidate method in the list by the special functions `callsame`, `callwith`, `nextsame`, and `nextwith`. The "same" variants reuse the original argument list passed to the current method, whereas the "with" variants allow a new argument list to be substituted for the rest of the candidates. The "call" variants dispatch to the rest of the candidates and return their values to the current method for subsequent processing, whereas while the "next" variants don't return, but merely defer to the rest of the candidate list:

23 This shows that it would be a whole lot easier to write the standard library as a library if there are some names that are reserved for use by the implementation. I suggest using a chosen non-letter such as ® as part of the identifier name.

24.1.3 callwith

This behaves like the Callable.callwith method, but automatically knows what the target Callable object is. See discussion under callsame on page 105.

24.1.4 context

24.1.5 leave

Same as context.leave. See page 119.

24.1.6 nextsame

This behaves like the Callable.nextsame method, but automatically knows what the target Callable object is. See discussion under callsame on page 105.

24.1.7 nextwith

This behaves like the Callable.nextwith method, but automatically knows what the target Callable object is. See discussion under callsame on page 105.

24.1.8 return

This returns from the Routine that lexically encloses the point of call at compile time.

> So does all of this apply to the general .leave function too? So the technical details should go on page 119.

It is equivalent to calling &?ROUTINE.leave with the same arguments. When called from a closure, it will fail if the &?ROUTINE noticed at cloning time is no longer in scope. The compiler may optimize it or implement it in some other way and does not necessarily call the leave function.

The argument list is preserved as a Capture object and responds to the left-hand Signature in a binding. This allows named return values if the caller expects one:

```
sub f () { return :x<1> }
sub g ($x) { print $x }

my $x := |(f);  # binds 1 to $x, via a named argument
g(|(f));        # prints 1, via a named argument
```

To return a literal Pair object, always put it in an additional set of parentheses:

```
return( (:x<1>), (:y<2>) ); # two positional Pair objects
```

Note that the postfix parentheses on the function call don't count as being "additional". However, as with any function, whitespace after the return keyword prevents that interpretation and turns it instead into a list operator:

```
return :x<1>, :y<2>; # two named arguments (if caller uses |)
return ( :x<1>, :y<2> ); # two positional Pair objects
```

If the function ends with an expression without an explicit return, that expression is also taken to be a Capture, just as if the expression were the argument to a return list operator (with whitespace):

```
sub f { :x<1> } # named-argument binding (if caller uses |)
sub f { (:x<1>) } # always just one positional Pair object
```

On the caller's end, the Capture is interpolated into any new argument list much like an array would be, that is, as an item in item context, and as a list in list context. This is the default behavior, but the caller may use prefix:<|> to inline the returned values as part of the new argument list. The caller may also bind the returned Capture directly.

If any function called as part of a return list asks what its context is, it will be told it was called in list context regardless of the eventual binding of the returned Capture.[24] If that is not the desired behavior you must coerce the call to an appropriate context, (or declare the return type of the function to perform such a coercion). In any event, such a function is called only once at the time the Capture object is generated, not when it is later bound (which could happen more than once).

So, what is the role of the returns and of (outer and inner return) types?

24.1.9 self

This is defined implicitly in every method or submethod to return the invocant. It may be called from the pseudo-assignments of the method's or submethod's argument defaults.

The return value has the same declared type as the class containing the method or submethod definition (note that this may be different than an explicitly declared invocant in the parameter list). It returns the object in item context.

24.1.10 temp

What about precedence? Check to see if it needs to be odd, thus not classified here.

Parses like:

```
sub temp (::T $lvalue is ref --> T)
```

This takes a modifiable lvalue as its argument, and returns the same lvalue, transparently. But as a side-effect it calls the .TEMP method on the argument's container, and appends returned closure to the enclosing block's LEAVE trait.

The line

```
temp $x = $newval;
```

24 This is quite different from Perl 5, where a return statement always propagates its caller's context to its own arguments.

Is approximately equivalent to

```
my $@nonce = $x.TEMP is leave {$_.()}
$x = $newval;
```

It is a macro, because the LEAVE trait must be set up at compile time, with the actual closure value found at run time. The behavior of a goto to run the temp function again without leaving and re-entering the block is unspecified.

24.1.11 want

Short for caller.want. See page 120. Note that this is not necessarily the same as context.want.

24.1.12 VAR

When applied to an item, redirects attention from the contents to the container. This basically suppresses the normal semantics of automatically referring to the contents when using an item value.

The semantics are truly built-in and cannot be explained in terms of anything else.

This might work as a method too.

24.2 Any (class)

Mutable

Perl 6 object (default parameter type, excludes Junction)

24.3 Array (class)

Mutable

does Positional

The various methods should be pushed up to a base class or rule.

24.3.1 any

Returns a junction of all the array's elements.

24.3.2 bytes

The total string length of an array's elements, in bytes.

Need to define "total string length" of an array.

24.3.3 chars

Other Str methods are available on Array. This was left out — not on purpose? Do we really

need to duplicate all the methods of strings? Why not just one function to get the concatenated string? So I think bytes, codes, and graphs and this one are up for appeal.

24.3.4 codes

The total string length of an array's elements, in code points.

Need to define "total string length" of an array.

24.3.5 elems

To get the number of elements in an array, use the `.elems` method.

24.3.6 graphs

The total string length of an array's elements, in graphemes.

Need to define "total string length" of an array.

24.3.7 push

24.4 AST

Quasiquoting returns one. Need standard definition so macros can be portable.

24.5 Bag

Immutable

Unordered collection of values that allows duplicates

24.6 bit (compact type)

24.7 Bit (class)

Immutable

Do we really need a class for this?

24.8 Blob (class)

Immutable

An undifferentiated mass of bits

24.9 Block (class)

Immutable

does Callable

A Block is a Code object that provide lexical scope to its contents, and various attributes attached to the Code object as described in this section. This is the most primitive concrete class that does Code in the P6 standard library.

24.9.1 leave

You can exit any labeled block early by saying, e.g.

```
MyLabel.leave(@results);
```

24.10 bool (compact type)

24.11 Bool (enum)

Immutable

Constants True and False, exported globally.

Is it really an enum? Are they constants of Bool or bool or both?

Should be an enum so autotyping of "but" works.

24.12 Buf and buf

Mutable

If a buf type is initialized with a Unicode string value, the string is decomposed into Unicode codepoints, and each codepoint shoved into an integer element. If the size of the buf type is not specified, it takes its length from the initializing string. If the size is specified, the initializing string is truncated or 0-padded as necessary. If a codepoint doesn't fit into a buf's integer type, a parse error is issued if this can be detected at compile time; otherwise a warning is issued at run time and the overflowed buffer element is filled with an appropriate replacement character, either `U+FFFD` (REPLACEMENT CHARACTER) if the element's integer type is at least 16 bits, or `U+007f` (DELETE) if the larger value would not fit. If any other conversion is desired, it must be specified explicitly. In particular, no conversion to UTF-8 or UTF-16 is attempted; that must be specified explicitly. (As it happens, conversion to a buf type based on 32-bit integers produces valid UTF-32 in the native endianness.)

A `Buf` is a stringish view of an array of integers, and has no Unicode or character properties without explicit conversion to some kind of `Str`. (A `buf` is the native counterpart.) Typically it's an array of bytes serving as a buffer. Bitwise operations on a `Buf` treat the entire buffer as a single large integer. Bitwise operations on a `Str` generally fail unless the `Str` in question can provide an abstract `Buf` interface somehow. Coercion to `Buf` should generally invalidate the `Str` interface. As a generic type `Buf` may be instantiated as (or bound to) any of `buf8`, `buf16`, or `buf32` (or to any type that provides the appropriate `Buf` interface), but when used to create a

buffer `Buf` defaults to `buf8`.

Unlike `Str` types, `Buf` types prefer to deal with integer string positions, and map these directly to the underlying compact array as indices. That is, these are not necessarily byte positions--an integer position just counts over the number of underlying positions, where one position means one cell of the underlying integer type. Builtin string operations on `Buf` types return integers and expect integers when dealing with positions. As a limiting case, `buf8` is just an old-school byte string, and the positions are byte positions. Note, though, that if you remap a section of `buf32` memory to be `buf8`, you'll have to multiply all your positions by 4.

24.13 Callable (role)

This role is notable as being the interface for variables defined using the & sigil.

24.13.1 (∞∞)

```
method postcircumfix:<( )> (|$args)
```

This is what is called when the ordinary "function call" notation is used on a scalar ($) or callable (&) value.

The implementation may check the suitability of the arguments and issue errors or warnings at compile time, if the signature of the Callable object is known at compile time. Note that the compiler may know what the signature of the Callable object must be without knowing the actual instance of Callable that will be involved.

24.13.2 arity

```
method arity()
of Int
```

The number of required parameters a subroutine has can be determined by calling its .arity method:

```
$args_required = &foo.arity;
```

This is the same as the method on the Signature. In fact, this should delegate to the signature of the block, and there should be .count as well (or neither).

24.13.3 callwith

```
method callwith (|$args)
```

Use of callwith allows the routine to be called without introducing an official CALLER frame.

24.13.4 signature

method signature ()
of Signature

Returns the Signature installed as the parameter list.

24.14 Capture (class)

Immutable

Function call arguments (right-hand side of a binding)

24.15 Class (class)

Mutable

Perl 6 standard class namespace

24.16 Code (class)

Immutable

Does Callable.

24.16.1 assuming

Every Code object has a .assuming method. This method does a partial binding of a set of arguments to a signature and returns a new function that takes only the remaining arguments.

```
my &textfrom := &substr.assuming(str=>$text, len=>Inf);
```

or equivalently:

```
my &textfrom := &substr.assuming(:str($text) :len(Inf));
```

or even:

```
my &textfrom := &substr.assuming:str($text):len(Inf);
```

It returns a Code object that implements the same behavior as the original subroutine, but has the values passed to .assuming already bound to the corresponding parameters:

```
$all  = textfrom(0);      # same as: $all  = substr($text,0,Inf);
$some = textfrom(50);     # same as: $some = substr($text,50,Inf);
$last = textfrom(-1);     # same as: $last = substr($text,-1,Inf);
```

The following paragraph really belongs under general information for specifying a variation of a multi by name, any time you want to refer to the Routine symbol.

To curry a particular multi variant, it may be necessary to specify the type for one or more of its parameters:

```
&woof ::= &bark:(Dog).assuming :pitch<low>;
&pine ::= &bark:(Tree).assuming :pitch<yes>;
```

This should be filed under that object type, explained as part of use, and cross-referenced from here.

The result of a use statement (so it's not a "statement" --JMD) is a (compile-time) object that also has a .assuming method, allowing the user to bind parameters in all the module's subroutines/methods/etc. simultaneously:

```
(use IO::Logging).assuming(logfile => ".log");
```

This special form should generally be restricted to named parameters.

24.16.2 labels

This returns a list of the labels in the block. It does not include in-line nested blocks; only labels that are directly part of the block.

```
if &?BLOCK.labels.any eq 'Foo' {...}
```

24.17 Comparable (role)

Types which support this role indicate that they are at least partly ordered. It means that the relational operators are defined, but does not say anything about the enumerable, continuous, or fully ordered.

In creating a class, including this role is a handy way to get all the relational operators defined in a standard way, while only having to write an implementation for cmp.

Note that the operators cmp, before, after, !before, !after are the "native" comparison operators of the values of a type. Using <=>, <, >, >=, <= are numeric and are defined similarly in the ComparableNumeric Role.

For types which are not fully ordered, you cannot assume that at least one of before, after, or eqv is true.

All orderable object types must support `+Inf` and `-Inf` values as special forms of the undefined value. It's an error, however, to attempt to store an infinite value into a native type that cannot support it

The syntax of "foo" min +Inf means that Inf must implicitly convert to the type. Based on mailing list posting 24-April-2008, I think that Inf as a literal will produce a unique type that converts to any Comparable type. So operators can code that Inf type directly as a signature, or compare the resulting value.

The role ought to automatically mix-in this conversion.

24.17.1 after, !after, before, !before

```
method infix:<after> (::?CLASS $right, *%adverbs)
of Bool
is inline
is export
```

similar for all 4 relationships

These are defined automatically by the Role to supply these operators in terms of the cmp function.

The functions return False if the comparison is NoRelation. Having it return False but Order::NoRelation would make it difficult for the optimizer to treat these as simple tests. If you want to know whether there is a relationship, use the cmp function directly. Normally if you want to know, for example, if one Type is a subtype of another, False is a good enough answer: you don't care whether it's false because they are ordered the other way around or because they are not related.

The implementation is equivalent to:

```
method infix:<before> (::?CLASS $right, *%adverbs --> Bool)
is export is inline
 {
 return .cmp($right, |%adverbs) == Order::Less;
 }

method infix:<!after> (::?CLASS $right, *%adverbs --> Bool)
is export is inline
 {
 return .cmp($right, |%adverbs) != Order::Greater;
 }

method infix:<after> (::?CLASS $right, *%adverbs --> Bool)
is export is inline
 {
 return .cmp($right, |%adverbs) == Order::Greater;
 }

method infix:<!before> (::?CLASS $right, *%adverbs --> Bool)
is export is inline
 {
 return .cmp($right, |%adverbs) != Order::Less;
 }
```

24.17.2 complete

```
method ^complete ()
of Bool
is qualified
```

The concrete type must supply this class method. It should return True if the ordering relationship between values of this type is complete. That is, cmp will never return Order::NoRelation.

24.17.3 cmp

```
multi method cmp (::?CLASS $right)
of Order
is export
```

Concrete types must define this function. The relational operators then work automatically.

It should return one of the Order enumeration values Order::Increase, Order::Same, Order::Decrease or Order::NoRelation.

```
multi method cmp (Infinity $right)
of Order
is export
multi method cmp (Infinity $right -->Order) is export
{
 return $right.negative ?? Order::Decrease !! Order::Increase;
 }
```

24.18 ComparableNumeric

Types which support this role indicate that they are at least partly ordered and use the numeric Perl comparison operators to indicate such ordering. It means that the numeric relational operators are defined, but does not say anything about being enumerable, continuous, or fully ordered.

In creating a class, including this role is a handy way to get all the relational operators defined in a standard way, while only having to write an implementation for <=>.

For types which are not fully ordered, you cannot assume that at least one of >, <, or == is true.

If you want to extend the set of commonly available operators, such as defining infix:<≤>, infix:<≠>, etc. as being synonymous with the usual spellings, do it in this role and they will be available to all the standard types.

24.18.1 < , > , <= , >=

```
method infix:{'<'} (::?CLASS $right)
of Bool
is inline
is export
```

similar for all 4 relationships

These are defined automatically by the Role to supply these operators in terms of the cmp function.

The functions return False if the comparison is NoRelation. Having it return False but Order::NoRelation would make it difficult for the optimizer to treat these as simple tests. If you want to know whether there is a relationship, use the cmp function directly.

The implementation is equivalent to:

```
method infix:{'<'} (::?CLASS $right --> Bool)
is export is inline
 {
 return .infix{'<=>'}($right) == Order::Less;
 }
method infix:{'<='} (::?CLASS $right --> Bool)
is export is inline
 {
 return .infix{'<=>'}($right) != Order::Greater;
 }
method infix:{'>'} (::?CLASS $right --> Bool)
is export is inline
 {
 return .infix{'<=>'}($right) == Order::Greater;
 }
method infix:{'>='} (::?CLASS $right --> Bool)
is export is inline
 {
 return .infix{'<=>'}($right) != Order::Less;
 }
```

24.18.2 complete

```
method ^complete ()
of Bool
is qualified
```

The concrete type must supply this class method. It should return True if the numeric ordereng relationship between values of this type is complete. That is, <=> will never return Order::NoRelation.

24.18.3 <=>

```
method infix{'<=>'} (::?CLASS $right)
of Order
is export
```

Concrete types must define this function. The relational operators then work automatically.

It should return one of the Order enumeration values Order::Increase, Order::Same, Order::Decrease or Order::NoRelation.

24.19 Context (role)

The S06 section “context and caller” describe the return object as having this role, but never names it as such.

24.19.1 context

The `.context` and `.caller` methods work the same as the functions except that they are relative to the context supplied as invocant.

Put the real description here, as the ambient pseudo-function context points here.

24.19.2 caller

The `.context` and `.caller` methods work the same as the functions except that they are relative to the context supplied as invocant.

Put the real description here, as the ambient pseudo-function context points here.

24.19.3 file

24.19.4 hints

The `.hints` method gives access to a snapshot of compiler symbols in effect at the point of the call when the call was originally compiled. (For instance, `caller.hints('&?ROUTINE')` will give you the caller's routine object.) Such values are always read-only, though in the case of some (like the caller's routine above) may return a fixed object that is nevertheless mutable.

24.19.5 inline

The `.inline` method says whether this block was entered implicitly by some surrounding control structure. Any time you invoke a block or routine explicitly with `.()` this is false. However, it is defined to be true for any block entered using dispatcher-level primitives such as `.callwith`, `.callsame`, `.nextwith`, or `.nextsame`.

24.19.6 leave

The `.leave` method can force an immediate return from the specified context.

24.19.7 line

24.19.8 my

The `.my` method provides access to the lexical namespace in effect at the given dynamic context's current position. It may be used to look up ordinary lexical variables in that lexical scope. It must not be used to change any lexical variable that is not marked as `context<rw>`.

24.19.9 package

24.19.10 want

I think this returns the signature of the return (inner or outer?), but might really interact with the caller's context. Need Larry to explain this whole feature better.

24.20

24.21 Complex (class)

Immutable

24.22 Encoding (class)

Not in S26 or S16 now. Presumably modeled after the encoding pragma/module in Perl 5. Someone needs to pull that in, and address Perl 6 issues and any changes based on hindsight.

Of particular importance is to document the supported encoding names, so that any standard program may rely on them.

Presumably, some will be mandatory, and others will be optional or available with installed support files.

24.23 Enumerated (role)

Does the Comparable role.

This is used to implement classes that are enumeration types.

24.24 Exception (class)

Immutable

24.25 Failure (role)

24.26 GLOBAL (package)

24.27 Grammar (class)

Mutable

Perl 6 pattern matching namespace

24.28 Hash (class)

Mutable

24.29 Infinite

Immutable.

This class holds values that represents the transfinite cardinals, including Inf and -Inf. Higher cardinals $\pm\aleph_n$ can be represented, but don't have any special meaning to the standard library.

Infinite values do have special meaning within the standard library, in that it converts to many different types implicitly, and any Comparable type must support comparisons against ±Inf even if the class doesn't support holding infinite values.

The term Inf generates an Infinite constant.

The Comparable role, and various other classes, supply multi functions to treat comparisons against Inf as larger (or smaller for -Inf) than any legal value.

Classes that support the concept of infinity, such as Num and Int, actually make it a special form of undef, easily constructed along the lines of Int but Inf. That is, the typed undefined protoobject with Inf added as a property.

This is true even for Num, as opposed to using the IEEE encoding of the native Inf value. This allows all types to treat Inf similarly to a Failure, which is especially handy in vector, parallel, and hyperoperators.

For types that support infinite values, Inf will convert implicitly, so you can write:

```
my Int $x = Inf;
my Num $y = -Inf;
```

and not need a different typed infinity for each type.

Types that don't support infinities will reject the assignment as a type mismatch. But you can still pass Inf, in its native Infinite type, as an argument to comparison functions to compare any Comparable type against infinity. However, you could not assign it to a typed variable:

```
my Str $s1 = -Inf;  # error
my $s2 = -Inf;  # forgo type information
```

```
my Str|Infinite $s3 = -Inf;  # this one works
my Str $s4 = Str but -Inf;  # assignment works, but subsequent lt/gt operator doesn't
```

The comparison of "xxx" gt $s2 works using MMD, finding a form of gt that operates between a Str and an Infinity.

```
multi sub infix:<gt> (Str $, Infinity $right ;; *%adverbs -->Bool)
{
 if $right.positive  return False;  # nothing is gt infinity
 else return True;  # everything is gt -Inf
 }
```

The comparison of "xxx" gt $s3 calls the same function.

But "xxx" gt $s4 will dispatch to the form that compares two Str objects, which will object to the undef value by failing.

24.29.1 Roles

does Comparable,

does ComparableNumeric

24.29.2 -

```
method prefix:<-> (-->::?CLASS)
is export
is inline
```

This reverses the sign of the value. The implementation is something like:

```
method prefix:<->(-->::CLASS) is export is inline
{
 return .new(cardinal=>- .cardinal);
 }
```

24.29.3 <=>, cmp

Between Infinity objects, the comparisons will compare the cardinal values.

Between Infinity objects and any Comparable or ComparableNumeric type (for <=>) a positive Infinity is greater than any non-Infinity value, and negative Infinity is less than any non-Infinity value. An Infinity that is neither positive nor negative will behave indeterminately.

```
multi method cmp (£ Comparable $ --> Order)
{
 return .positive ?? Order::Decrease !! Order::Increase;
 }
```

The method that takes Infinity as the parameter is part of the Comparable role.

The implementation is similar for <=>.

24.29.4 cardinal

```
method cardinal (-->Int)
is rw
is inline
```

This is a r/w accessor for the cardinality of the transfinite number. Normal Inf has a cardinality of 1, and -Inf has a cardinality of -1. The sign indicates the sign direction of the infinity, and the magnitude represents different transfinite sets.

The standard library puts no interpretation on these values, other than for the sign. The intent is that Aleph-null ($\aleph_0$) is cardinal 1, $\aleph_1$ is cardinal 2, and $\aleph_n$ is cardinal n+1. A value of 0 is tolerated, but is not specifically intended to mean anything and may behave erratically as either +Inf or -Inf in the Infinity support in the standard library.

24.29.5 negative

```
method negative (-->Bool)
is inline
```

This returns True if the infinity is in the negative direction. Given Infinity $i, the tests $i.negative, $i <= -Inf, and $i.cardinal < 0 are all equivalent. Using this method is intended to be a particularly efficient way to make this determination.

24.29.6 new

```
method new (Int $cardinal = 1 --> ::?CLASS)
```

The standard method new. The only parameter available is the cardinal.

24.29.7 positive

```
method positive (-->Bool)
is inline
```

This returns True if the infinity is in the positive direction. Given Infinity $i, the tests $i.positive, $i >= Inf, and $i.cardinal > 0 are all equivalent. Using this method is intended to be a particularly efficient way to make this determination.

24.30 int (compact type)

int is the same size as one of the numbered int variations. It is the size of the largest integer type that runs at full speed. The conformance statement of an implementation must state what that size is.

Note that int types support native arithmetic model of the CPU, and is not necessary two's complement. The conformance statement of an implementation must state whether it is 2's complement or not, and what the behavior is otherwise.

24.31 Integral (role)

Proposed, strawman. To organize the supplied operators and allow for flexibility of generics with constraints, we need to separate out the built-in types into more primordial categories, and can then define the functions that work across several types on these roles.

Integral is for any type that works with integers, including all the flavors of Int. Generics defined on this type can operate uniformly on all the integer-like types.

24.31.1

24.32 Int (class)

Immutable

Int automatically supports promotion to arbitrary precision, as well as holding Inf and NaN values. Note that Int assumes 2's complement arithmetic, so +^1 == -2 is guaranteed.

> How do we name the Inf and NaN values? Different for Int and Num types like in C#? I think it should figure it out for itself like 0 can. Also need separate positive and negative infinities.

24.32.1 operator div

```
multi sub infix:<div> (Int $top, Int $bottom --> Rat)
is inline
```

The div operator applied between two Ints will produce a ratio.

This form will handle div applied between sized ints and uints of any type, and always produces a Rat. The Rat type doesn't have a native type counterpart, so there is no need to match the argument sizes.

It is unspecified whether there are additional multi forms defined between any of the sized int and uint types, or whether this form relies on automatic promotion of the arguments. An implementation can supply those forms or not as best fits its efficiency concerns.

The implementation may inline and optimize this call at compile time to produce an in-place Rat literal if it can.

24.32.2 abs

```
our ::?CLASS multi method abs ( $x: )
is export
is inline
```

Absolute value. Equivalent to

```
multi method abs (-->::?CLASS)
is export is inline
 {
 return self >= 0 ?? self !! -self;
 }
```

Many functions are in both Int and Num, and many can have common polymorphic implementations, like this one. I think they can be abstracted out to a Role.

24.33 IO (role)

Mutable

Perl filehandle

does KitchenSink (Note that an `IO` object used as a sink will force eager evaluation on its pipeline, so the next statement is guaranteed not to run till the file is closed.)

24.33.1 close

24.34 Junction (class)

Immutable

Set with additional behaviors

Is this related to the junction feature? So do these need to be parameterized?

24.35 KeyBag (class)

Mutable

KeyHash of UInt (does Bag in list/array context)

A `KeyBag` is a `KeyHash` of `UInt` with default of 0. If you use the `Hash` interface and increment an element of a `KeyBag` its value is increased by one (creating the element if it doesn't exist already). If you decrement the element the value is decreased by one; if the value goes to 0 the element is automatically deleted. An attempt to decrement a non-existing value results in a `Failure` value. When not used as a `Hash` (that is, when used as an `Array` or list or `Bag` object) a `KeyBag` behaves as a `Bag` of its keys, with each key replicated the number of times specified by its corresponding value. (Use `.kv` or `.pairs` to suppress this behavior in list context.)

24.36 KeyHash (class)

Mutable

Perl hash that autodeletes values matching default

A `KeyHash` differs from a normal `Hash` in how it handles default values. If the value of a `KeyHash` element is set to the default value for the `KeyHash`, the element is deleted. If undeclared, the default default for a `KeyHash` is 0 for numeric types,

`False` for boolean types, and the null string for string and buffer types. A `KeyHash` of a `Object` type defaults to the undefined prototype for that type. More generally, the default default is whatever defined value an `undef` would convert to for that value type. A `KeyHash` of `Scalar` deletes elements that go to either 0 or the null string. A `KeyHash` also autodeletes keys for normal undef values (that is, those undefined values that do not contain an unthrown exception).

24.37 KeySet (class)

Mutable

KeyHash of Bool (does Set in list/array context)

A `KeySet` is a `KeyHash` of booleans with a default of `False`. If you use the `Hash` interface and increment an element of a `KeySet` its value becomes true (creating the element if it doesn't exist already). If you decrement the element it becomes false and is automatically deleted. Decrementing a non-existing value results in a `False` value. Incrementing an existing value results in `True`. When not used as a `Hash` (that is, when used as an `Array` or list or `Set` object) a `KeySet` behaves as a `Set` of its keys. (Since the only possible value of a `KeySet` is the `True` value, it need not be represented in the actual implementation with any bits at all.)

24.38 KitchenSink (role)

Mentioned in S06 under Feed Operators.

24.38.1 clear

24.38.2 push

24.39 List (class)

Immutable

Lazy Perl list (composed of immutables and iterators)

Contrast with Seq.

24.40 Macro (class)

Mutable

Perl compile-time subroutine

24.41 Mapping (class)

Immutable

Set of Pairs with no duplicate keys

24.42 Match (class)

Mutable

Perl match, usually produced by applying a pattern

24.43 Metaobject (role) aka Type?

The thing returned by HOW does not have to be any specific type, but can be an instance designed for whatever kind of object system is desired. To that end, this is defined as an interface, not an actual type.

> This might be better called Type. Then again, most generic code doesn't need to know what it is called.

24.43.1 convert:<Str>

```
method convert:<Str> ()
is implicit
```

Stringification will produce a string of the general format "Metaobject$s" where $s is an implementation-dependent message.

For Metaobjects that control standard Perl 6 classes, the message will contain at least a unique ID number for this Metaclass value, the name of the class it is associated with, and the names and ID numbers of any mix-in properties, and indications as to whether this is not an "ordinary" object but a protoobject or other special form.

For Metaobjects that are proxies or wrappers for foreign classes, the message should return some suitable indication to that effect.

Metaobjects created for any other purpose should continue in this tradition.

For example,

```
say "10".HOW;
$x = 0.0 but True;   say $x.HOW;
$x = Dog.new("fido");   say $x.HOW;
say Dog.HOW;
$x = doesntwork(1);   say $x.HOW;
```

will print results similar to:

```
Metaobject #5 for Str
Metaobject #18 for Num#3 but Bool#7
Metaobject #242 for Dog
Metaobject #243 for Dog#242 protoobject
Metaobject #87 for Num#3 but Failure#6
```

Note that except for the initial word, the exact form of the string is implementation dependent and meant for human reading during troubleshooting, not for parsing. If you want to explore the Metaobject programmatically, use the methods such as

name and buts that return information in well-defined forms.

24.43.2 attributes

Not autodelegated from Object

24.43.3 bless

24.43.4 buts

method buts (-->Metaobject @)

For Metaobjects that control objects with added properties, this lists all the primary Metaclasses that are involved. The first entry will be the Metaobject of the main type that had properties added to it. Additional entries list Mataobjects of each mix-in.

This is flattened, with the primary being the original class, not a Metaobject with one less property.

For Metaobjects that are some kind of proxy or wrapper, the first entry indicates the type that this is proxying for, and subsequent entries are types that control the proxy object itself.

Other kinds of Metaobjects are expected to use this list in a similar way: list everything that was “bundled together” to form this Metaobject.

24.43.5 can

According to S12: Unlike in Perl 5 where `.can` returns a single `Code` object, Perl 6's version of `.HOW.can` returns a "WALK" iterator for a set of routines that match the name, including all autoloaded and wildcarded possibilities. In particular, `.can` interrogates any class package's `CANDO` method for names that are to be considered autoloadable methods in the class, even if they haven't been declared yet. Role composition sometimes relies on this ability to determine whether a superclass supplies a method of a particular name if it's required and hasn't been supplied by the class or one of its roles.

24.43.6 does

24.43.7 isa

24.43.8 methods

Not autodelegated from Object

method methods (*TBD*)
of Routine @

In S12, "The `.methods` method has a selector parameter that lets you specify whether you want to see a flattened or hierarchical view, whether you're interested in private methods, and so forth."

I put the various methods shown as part of the "method descriptor" on the Routine class, and figure this returns a list of Routine objects.

24.43.9 name

```
method name (-->Str,
        Bool :$qualified,
        Bool :$strict
        )
```

Returns the name of the class. The returned name does not include a :: sigil.

This returns the name of the class.

If the :qualified adverb is used, the fully-qualified name will be returned, tracing back nested scopes back to (but not including) the global scope. E.g. a return value might be "Package::Outer::Inner::Classname".

If the :strict adverb is used, the name will be a name legal to use in source text. Any non-identifier names encountered will use suitable syntax to express. If there are anonymous items, the function fails (*Which error?*).

If the :strict adverb is not used, anonymous parts will be replaced by explanatory text, and non-identifier name parts will be included as-is, except for a part that contains the sequence "::" (that is, not in the role of a part separator). The abnormal use of "::" will be explained or escaped in a manner suitable for human readers.

> This is my best model for how all the various classes/roles that should have a name method ought to work. Goes with names below.

24.43.10 names

```
method names ()
of Str @
```

This returns all the parts of the qualified name, in a list. The last entry is the class name itself. An anonymous name will have an undef entry. The name parts have no special processing with regard to being legal identifiers or not.

24.44 Method (class)

Mutable

> Is this derived from Sub?

24.45 Module

Mutable

Perl 6 standard namespace

24.46 MY (pseudo-package)

24.47 Num (class)

Immutable

Num must support the largest native floating point format that runs at full speed, and is the same size as num. The conformance statement of an implementation must state which size this is.

Within a lexical scope, pragmas may specify the nature of temporary values, and how floating point is to behave under various circumstances. All IEEE modes must be lexically available via pragma except in cases where that would entail heroic efforts to bypass a braindead platform.

The default floating-point modes do not throw exceptions but rather propagate Inf and NaN. The boxed object types may carry more detailed information on where overflow or underflow occurred. Numerics in Perl are not designed to give the identical answer everywhere. They are designed to give the typical programmer the tools to achieve a good enough answer most of the time. (Really good programmers may occasionally do even better.) Mostly this just involves using enough bits that the stupidities of the algorithm don't matter much.

24.48 Object (class)

Mutable

> S02 states, "Perl 6 object (either Any or Junction)". Is this really a Role used as a wildcard, and distinct from the base class mentioned here? Object allows more stuff than Any, so is that a base/derived relationship, a subtype, constraint, or what?

The implementation may assume that all classes are *consistent* with Object. Object is a class, not a role, because it is all about physical storage management. If you want a class to do things differently, it can override methods like BUILDALL. But the implementation may have some deep magic involved with Object as the ultimate base class, that cannot be replaced. That is, any class package "isa" Object.

24.48.1 BUILD

24.48.2 BUILDALL

24.48.3 CANDO

24.48.4 clone

24.48.5 DESTROY

24.48.6 DESTROYALL

24.48.7 fmt

To get a formatted representation of any scalar value, use the `.fmt('%03d')` method to do an implicit `sprintf` on the value.

To format an array value separated by commas, supply a second argument: `.fmt('%03d', ', ')`. To format a hash value or list of pairs, include formats for both key and value in the first string: `.fmt('%s: %s', "\n")`.

24.48.8 HOW

our Metaobject method HOW ()

Every object supports a HOW function/method that returns the metaclass instance managing it, regardless of whether the object is defined:

```
'x'.HOW.methods;   # get available methods for strings
Str.HOW.methods;   # same thing with the prototype object Str
HOW(Str).methods;  # same thing as function call
'x'.methods;       # this is likely an error - not a meta object
Str.methods;       # same thing
```

(For a prototype system (a non-class-based object system), all objects are merely managed by the same meta object.)

The use of the ^ metasyntax is equivalent to to .HOW.

```
MyClass.HOW.methods();    # get the method list of MyClass
^MyClass.methods();       # get the method list of MyClass
MyClass.^methods();       # get the method list of MyClass
$obj.HOW.methods();
$obj.^methods();
```

24.48.9 new

24.48.10 perl

To get a Perlish representation of any object, use the `.perl` method. Like the `Data::Dumper` module in Perl 5, the `.perl` method will put quotes around strings, square brackets around list values, curlies around hash values, constructors around objects, etc., so that Perl can evaluate the result back to the same object.

24.48.11 WALK

According to S12:

```
my @candidates := $object.WALK(:name<foo>, :breadth, :omit($?CLASS));
$object.*@candidates(@args);
```

The `WALK` method takes these arguments:

```
:canonical      # canonical dispatch order
:ascendant      # most-derived first, like destruction order
:descendant     # least-derived first, like construction order
:preorder       # like Perl 5 dispatch
:breadth        # like multi dispatch

:super              # only immediate parent classes
:name<name>         # only classes containing named method declaration
:omit(Selector)     # only classes that don't match selector
:include(Selector)  # only classes that match selector
```

24.48.12 WHAT

24.48.13 WHEN

24.48.14 WHENCE

Returns the autovivification closure for the actual type.

Also can be used like this:

```
Dog but WHENCE({ :name<Fido> })
```

24.48.15 WHICH

Some object types can behave as value types. Every object can produce a "WHICH" value that uniquely identifies the object for hashing and other value-based comparisons. Normal objects just use their address in memory, but if a class wishes to behave as a value type, it can define a `.WHICH` method that makes different

objects look like the same object if they happen to have the same contents.

24.48.16 WHO

24.49 Ordinal (role)

A common interface for all types that are ordinal. Especially useful as a type constraint on generic type parameters, and for automatically implementing some operators if you define a new type that has Ordinal characteristics. That is, just define predecessor and successor and get all the pre/post operators defined for you.

Ordinal is also Ordered.

24.49.1 ++

```
method postfix:<++> ()
of ::?CLASS
is export
method prefix:<++> ()
of ::?CLASS
is rw
is export
```

Defined as equivalent to:

```
method postfix:<++> (-->::?CLASS)
is export
 {
 my ::?CLASS $temp = self;
 self =. successor;
 return $temp;
 }

method prefix:<++> (-->::?CLASS)
is export
is rw
 {
 self =.successor;
 return self;
 }
```

Note in particular the behavior if successor fails and failure is set to not throw: in the postfix form, the Failure object is stored as the new value of the object, and the old value returned from the function.

Note that the prefix form is an lvalue and the postfix form is not.

24.49.2 predecessor

```
method predecessor ()
of ::?CLASS
is inline
```

Returns the previous element. Fails if there is no previous.

Define which exception.

Concrete classes must provide this function.

24.49.3 strict

```
method ^strict ()
of Bool
is qualified
```

Invented "is qualified". Put in the docs and describe.

Real classes can be very useful without being mathematically strict about the meaning of "ordinal". For example, the Str class has a string form of increment and decrement, but calling successor repeatedly will not generate all possible strings, and I'm not even sure whether predecessor always is the exact inverse of successor.

Templates that use Ordinal as a type constraint can check for ^Ordinal::strict at run-time or as a Subtype constraint if they plan on making such assumptions. Such templates would fail, until the known class (such as Str) is explicitly programmed in as a special case.

Concrete classes must provide this function.

24.49.4 successor

```
method successor ()
of ::?CLASS
is inline
```

Returns the next element. Fails if there is no next.

Define which exception.

Concrete classes must provide this function.

24.50 Order (enum)

Contains enumeration values Order::Increase, Order::Same, Order::Decrease or Order::NoRelation.

Numifies to 1, 0, -1, and undef.

(should it be undef or Nan?)

Since comparing these enumeration constants is necessary in the use of the general

relational operators, it is guaranteed that infix:<eqv>(Order,Order) is directly implemented and does not defer to a more generic form. Normally, concrete classes need only implement the cmp operator and all the others have generic forms that call that.

24.51 P6opaque (class)

24.52 Package (class)

Mutable

does Tieable

Perl 5 compatible namespace

but, S03 says "Looking at it from the other direction, classes and modules that wish to be bound to a global package name must be able to do the `Package` role."

24.53 Pair (class)

Immutable

A single key-to-value association

Since Hashes can be strongly typed, are there different kinds of Pairs too?

24.54 Range (class)

Immutable

A pair of Ordered endpoints; gens immutables when iterated

24.55 Rat (class)

Immutable

`Rat` supports arbitrary precision rational arithmetic. However, dividing two `Int` objects using `infix:</>` produces a fraction of `Num` type, not a ratio. You can produce a ratio by using `infix:<div>` on two integers instead.

24.56 Regex (class)

Immutable, Mutable CONTRADICTION IN S02

24.57 Role (class)

Mutable

Perl 6 standard generic interface/implementation

24.58 Routine (class)

Mutable

does Code

Base class for all wrappable executable objects

24.58.1 as

the coercion type of the routine.

> Need to decide how far to push this up in the hierarchy.

24.58.2 do

The Routine's body.

> So is this a block? Which of the other functions here don't apply to blocks?

24.58.3 multi

```
method multi ()
of Bool
```

Whether duplicate names are allowed.

> Need to decide how far to push this up in the hierarchy.

24.58.4 name

The name of the current routine.

If the current routine is a multi, it returns the unambiguous long name.

If the current routine does not have a name, it returns undef.

If the current routine has a name that is not an identifier, it returns the normalized string form.

> Where I'm going with this is the general idea that this can always be used to write source code for a call to that routine. But, the normalized form of "circumfix:<< >>" causes a problem.
>
> At least arrange so this always matches the string you would need to look up a function symbolically.

The outermost routine at a file-scoped compilation unit is always named &MAIN in the file's package.

24.58.5 of

the outer return type of the routine.

Need to decide how far to push this up in the hierarchy.

24.58.6 signature

Inherited from Code

~~24.58.7 unwrap~~

I changed unwrap to a method on the returned handle object.

24.58.8 wrap

method wrap (&block)
is RestoreHandle

Every Routine object has a .wrap method. This method expects a single Code argument. Within the dynamic scope of the code, the special callsame, callwith, nextsame and nextwith functions will invoke the original routine, but do not introduce an official CALLER frame[25]:

```
sub thermo ($t) {...}   # set temperature in Celsius, returns old value

# Add a wrapper to convert from Fahrenheit...
$handle = &thermo.wrap( { callwith( ($^t-32)/1.8 ) } );
```

The callwith function lets you pass your own arguments to the wrapped function. The callsame function takes no argument; it implicitly passes the original argument list through unchanged.

What about type checking? The signature of the block must match or be fixed up somehow. Sounds the same as the conformance needed for virtual overrides.

The call to .wrap replaces the original Routine with the Code argument, and arranges that any call to callsame, callwith, nextsame or nextwith invokes the previous version of the routine. In other words, the call to .wrap has more or less the same effect as:

```
&old_thermo := &thermo;
&thermo = sub ($t) { old_thermo( ($t-32)/1.8 ) }
```

Signature type checking on the binding — need to cover this on the section on value types and binding.

The container object &thermo is updated in-place, so &thermo.WHICH stays the same after the .wrap.

The call to .wrap returns a unique handle that can later be used to undo the wrapping:

```
$handle.restore
```

This does not affect any other wrappings placed to the routine. That is, you can

25 That is, it is considered inline execution. See page 119.

remove a wrapper from the middle of a set of nested wrappings:

```
sub foo ($x) { say "original $x" }

my $h1 = &foo.wrap( {say "one before"; callsame; say "one after" } );
my $h2 = &foo.wrap( {say "two before"; callwith($^t + 1); say "two after" } );
my $h3 = &foo.wrap( {say "three before"; callsame; say "three after" } );

foo(42);
$h2.restore;
foo(42);
```

This will make three successive wrappings, and then the call produces the output:

```
three before
two before
one before
original 43
one after
two after
three after
```

Then the middle wrapping is removed, and foo is called again to produce this output:

```
three before
one before
original 42
one after
three after
```

The wrapper is not required to call the original routine; it can call another Code object, and still hide the wrapper by using the callwith method:

```
# This allows thermo to detect the wrapper
&thermo.wrap( sub (|$args) { other_thermo(|$args) } );
# Using callwith rather than a normal call hides the wrapper
&thermo.wrap( sub (|$args) { &other_thermo.callwith(|$args) } );
```

or more briefly:

```
&thermo.wrap( { &other_thermo.callsame } );
```

The non-method forms of callsame, callwith, nextsame, and nextwith only work inside wrappers. See page 105.

Since the method versions of `callsame`, `callwith`, `nextsame`, and `nextwith` specify an explicit destination, their semantics do not change outside of wrappers. However, the corresponding functions have no explicit destination, so instead they implicitly call the next-most-likely method or multi-sub; see S12 for details.

As with any return value, you may capture the returned `Capture` of `call` by binding:

```
my |$retval := callwith(|$args);
... # postprocessing
return |$retval;
```

Alternately, you may prevent any return at all by using the variants `nextsame` and `nextwith`. Arguments are passed just as with `callsame` and `callwith`, but a tail call is explicitly enforced; any code following the call will be unreached, as if a return had been executed there before calling into the destination routine.

Within an ordinary method dispatch these functions treat the rest of the dispatcher's candidate list as the wrapped function, which generally works out to calling the same method in one of our parent (or older sibling) classes. Likewise within a multiple dispatch the current routine may defer to candidates further down the candidate list. Although not necessarily related by a class hierarchy, such later candidates are considered more generic and hence likelier to be able to handle various unforeseen conditions (perhaps).

24.58.8.1 usage notes

Informative

The entire argument list may be captured by binding to a Capture parameter. It can then be passed to callwith using that name:

```
# Double the return value for &thermo
&thermo.wrap( -> |$args { callwith(|$args) * 2 } );
```

In this case only the return value is changed. That example could have used callsame.

24.59 Scalar (class)

Mutable

24.60 Seq (class)

Immutable

Completely evaluated (hence immutable) sequence

Contrast with List.

Seq and List need a common Role?

24.61 Set (class)

Immutable

Unordered collection of values that allows no duplicates

24.62 Signature

Immutable

Function parameters (left-hand side of a binding)

24.62.1 arity

method arity ()
of Int

Indicates the minimum number of positional parameters necessary to bind to this signature.

24.62.2 count

method count ()
of Int

Indicates the maximum number of positional parameters that can bind to this signature, including optional parameters. It includes *$ parameters but does not consider *@ (so it doesn't read Inf).

24.62.3 item

method item ()
of Bool

$_.item is equivalent to $_ ~~ :($).

The description in S06 is not clear: is it the smartmatch exactly the same or does it not distinghish lvalue?

24.62.4 list

method list ()
of Bool

$_.list is equivalent to $_ ~~ :(*@).

24.62.5 rw

method rw ()
of Bool

$_.list is equivalent to $_ ~~ :( ).

This is not shown in S04. Need to fill this in after understanding the smartmatch.

24.62.6 void

method void ()
of Bool

$_.list is equivalent to $_ ~~ :( ).

This is not shown in S04. Need to fill this in after understanding the smartmatch.

24.63 Str (class)

Immutable

A Str object has a current encoding. That needs to be fleshed out.

Text to be processed yet:

A `Str` is a Unicode string object. There is no corresponding native `str` type. However, since a `Str` object may fill multiple roles, we say that a `Str` keeps track of its minimum and maximum Unicode abstraction levels, and plays along nicely with the current lexical scope's idea of the ideal character, whether that is bytes, codepoints, graphemes, or characters in some language. For all builtin operations, all `Str` positions are reported as position objects, not integers.

If you use integers as arguments where position objects are expected, it will be assumed that you mean the units of the current lexically scoped Unicode abstraction level. (Which defaults to graphemes.) Otherwise you'll need to coerce to the proper units:

```
substr($string, 42.as(Bytes), 1.as(ArabicChars))
```

Of course, such a dimensional number will fail if used on a string that doesn't provide the appropriate abstraction level.

If a `StrPos` or `StrLen` is forced into a numeric context, it will assume the units of the current Unicode abstraction level. It is erroneous to pass such a non-dimensional number to a routine that would interpret it with the wrong units.

Informative

Implementation note: since Perl 6 mandates that the default Unicode processing level must view graphemes as the fundamental unit rather than codepoints, this has some implications regarding efficient implementation. It is suggested that all graphemes be translated on input to a unique grapheme numbers and represented as integers within some kind of uniform array for fast substr access. For those graphemes that have a precomposed form, use of that codepoint is suggested. (Note that this means Latin-1 can still be represented internally with 8-bit integers.)

For graphemes that have no precomposed form, a temporary private id should be assigned that uniquely identifies the grapheme. If such ids are assigned consistently thoughout the process, comparison of two graphemes is no more difficult than the comparison of two integers, and comparison of base characters no more difficult than a direct lookup into the id-to-NFD table.

Obviously, any temporary grapheme ids must be translated back to some universal form (such as NFD) on output, and normal precomposed graphemes may turn into either NFC or NFD forms depending on the desired output. Maintaining a particular grapheme/id mapping over the life of the process may have some GC implications for long-running processes, but most processes will likely see a limited number of

non-precomposed graphemes.

If the program has a scope that wants a codepoint view rather than a grapheme view, the string visible to that lexical scope must also be translated to universal form, just as with output translation. Alternately, the temporary grapheme ids may be hidden behind an abstraction layer. In any case, codepoint scope should never see any temporary grapheme ids. (The lexical codepoint declaration should probably specify which normalization form it prefers to view strings under. Such a declaration could be applied to input translation as well.)

24.63.1 Str::Canonicalization (type)

This describes the Unicode canonicalization. It has not been designed yet: is it an enumerated type, or something that can be extended by the user? Perhaps a list of independent attributes that can be set independently such as case folding, precomposed vs composite chars, etc.

24.63.2 bytes

```
method bytes (Encoding :$encoding = .encoding)
of Int
```

Returns the number of bytes in the encoded representation of the string.

Am I promising that if you derive from Str and override the .encoding method to return something other than what was last set as the encoding, it will work? That is, the implementation is prohibited from using self!encoding instead to speed it up. Or is a much longer-winded explanation needed for what the implementation is allowed to do?

24.63.3 chars

Chars in some language. Needs design work.

24.63.4 codes

```
method codes (Canonicalization :?canonicalization = .canonicalization)
of Int
```

Returns the number of code points in the string.

24.63.5 graphs

24.64 StrPos (class)

These `StrPos` objects point into a particular string at a particular location independent of abstraction level, either by tracking the string and position directly, or by generating an abstraction-level independent representation of the offset from

the beginning of the string that will give the same results if applied to the same string in any context. This is assuming the string isn't modified in the meanwhile; a `StrPos` is not a "marker" and is not required to follow changes to a mutable string. For instance, if you ask for the positions of matches done by a substitution, the answers are reported in terms of the original string (which may now be inaccessible!), not as positions within the modified string. (However, if you use `.pos` on the modified string, it will report the position of the end of the substitution in terms of the new string.)

The subtraction of two `StrPos` objects gives a `StrLen` object, which is also not an integer, because the string between two positions also has multiple integer interpretations depending on the units. A given `StrLen` may know that it represents 18 bytes, 7 codepoints, 3 graphemes, and 1 letter in Malayalam, but it might only know this lazily because it actually just hangs onto the two `StrPos` endpoints within the string that in turn may or may not just lazily point into the string. (The lazy implementation of `StrLen` is much like a `Range` object in that respect.)

24.65 Sub (class)

Mutable

is Routine

Perl subroutine

24.66 Submethod (class)

Mutable

24.67 SUPER (pseudo-package)

24.68 Tie (module)

Misc. note: storage classes have a TEMP method.

24.68.1 Tie::Array (role)

24.68.2 Tie::Hash (role)

24.68.3 Tie::Scalar (role)

24.69 Whatever (class)

```
class Whatever
is Any
```

This class is derived from Whatever, and behaves as a singleton. All instances of

Whatever compare as the same value. That is, WHICH always returns the same value.

It exists to support the * term. Operators and functions can detect the use of the *"whatever" by sensing that an object of type Whatever was used as an argument. This is easy for multi functions to dispatch on, or any code may check for type membership of an argument at run time.

```
multi sub foo (Int $x) { ... }
multi sub foo (Whatever $x) { ... }

sub bar (Int $x, Int|Whatever $y)
of Int
 {
 if $y.^does(Whatever) {
   # handle the 'whatever' case
   }
 else {
   my Int $yy = y;  # enable static typing by removing ambiguity
   # handle the regular case, using $yy instead of $y
   }
 }
```

A call to foo(*) will dispatch to the second form.

A call to bar(5,*) will handle the "whatever" within the function by checking the type of the argument.

24.69.1 multidimensional

method multidimensional (-->Bool)

Since all instances of Whatever are the same, comparing them doesn't do anything useful. However, there are two forms of the "whatever" term: * and **. Code can distinguish them by calling this method.