Nim Manual

Search:

Authors: Andreas Rumpf, Zahary Karadjov

Version: 0.17.2

"Complexity" seems to be a lot like "energy": you can transfer it from the end user to one/some of the other players, but the total amount seems to remain pretty much constant for a given task. -- Ran

About this document

Note: This document is a draft! Several of Nim's features may need more precise wording. This manual is constantly evolving until the 1.0 release and is not to be considered as the final proper specification.

This document describes the lexis, the syntax, and the semantics of Nim.

The language constructs are explained using an extended BNF, in which `(a)*` means 0 or more `a`'s, `a+` means 1 or more `a`'s, and `(a)?` means an optional *a*. Parentheses may be used to group elements.

`&` is the lookahead operator; `&a` means that an `a` is expected but not consumed. It will be consumed in the following rule.

The `|`, `/` symbols are used to mark alternatives and have the lowest precedence. `/` is the ordered choice that requires the parser to try the alternatives in the given order. `/` is often used to ensure the grammar is not ambiguous.

Non-terminals start with a lowercase letter, abstract terminal symbols are in UPPERCASE. Verbatim terminal symbols (including keywords) are quoted with `'`. An example:

```
ifStmt = 'if' expr ':' stmts ('elif' expr ':' stmts)* ('else' stmts)?
```

The binary `^*` operator is used as a shorthand for 0 or more occurrences separated by its second argument; likewise `^+` means 1 or more occurrences: `a ^+ b` is short for `a (b a)*` and `a ^* b` is short for `(a (b a)*)?`. Example:

```
arrayConstructor = '[' expr ^* ',' ']'
```

Other parts of Nim - like scoping rules or runtime semantics are only described in the, more easily comprehensible, informal manner for now.

Definitions

A Nim program specifies a computation that acts on a memory consisting of components called locations. A variable is basically a name for a location. Each variable and location is of a certain type. The variable's type is called static type, the location's type is called dynamic type. If the static type is not the same as the dynamic type, it is a super-type or subtype of the dynamic type.

An identifier is a symbol declared as a name for a variable, type, procedure, etc. The region of the program over which a declaration applies is called the scope of the declaration. Scopes can be nested. The meaning of an identifier is determined by the smallest enclosing scope in which the identifier is declared unless overloading resolution rules suggest otherwise.

An expression specifies a computation that produces a value or location. Expressions that produce locations are called l-values. An l-value can denote either a location or the value the location contains, depending on the context. Expressions whose values can be determined statically are called constant expressions; they are never l-values.

A static error is an error that the implementation detects before program execution. Unless explicitly classified, an error is a static error.

A checked runtime error is an error that the implementation detects and reports at runtime. The method for reporting such errors is via *raising exceptions* or *dying with a fatal error*. However, the implementation provides a means to disable these runtime checks. See the section pragmas for details.

Whether a checked runtime error results in an exception or in a fatal error at runtime is implementation specific. Thus the following program is always invalid:

```
var a: array[0..1, char]
let i = 5
try:
  a[i] = 'N'
except IndexError:
  echo "invalid index"
```

An unchecked runtime error is an error that is not guaranteed to be detected, and can cause the subsequent behavior of the computation to be arbitrary. Unchecked runtime errors cannot occur if only safe language features are used.

Lexical Analysis

Encoding

All Nim source files are in the UTF-8 encoding (or its ASCII subset). Other encodings are not supported. Any of the standard platform line termination sequences can be used - the Unix form using ASCII LF (linefeed), the Windows form using the ASCII sequence CR LF (return followed by linefeed), or the old Macintosh form using the ASCII CR (return) character. All of these forms can be used equally, regardless of platform.

Indentation

Nim's standard grammar describes an indentation sensitive language. This means that all the control structures are recognized by indentation. Indentation consists only of spaces; tabulators are not allowed.

The indentation handling is implemented as follows: The lexer annotates the following token with the preceding number of spaces; indentation is not a separate token. This trick allows parsing of Nim with only 1 token of lookahead.

The parser uses a stack of indentation levels: the stack consists of integers counting the spaces. The indentation information is queried at strategic places in the parser but ignored otherwise: The pseudo terminal **IND{>}** denotes an indentation that consists of more spaces than the entry at the top of the stack; **IND{=}** an indentation that has the same number of spaces. **DED** is another pseudo terminal that describes the *action* of popping a value from the stack, **IND{>}** then implies to push onto the stack.

With this notation we can now easily define the core of the grammar: A block of statements (simplified example):

```
ifStmt = 'if' expr ':' stmt
         (IND{=} 'elif' expr ':' stmt)*
         (IND{=} 'else' ':' stmt)?

simpleStmt = ifStmt / ...

stmt = IND{>} stmt ^+ IND{=} DED  # list of statements
     / simpleStmt                 # or a simple statement
```

Comments

Comments start anywhere outside a string or character literal with the hash character #. Comments consist of a concatenation of comment pieces. A comment piece starts with # and runs until the end of the line. The end of line characters belong to the piece. If the next line only consists of a comment piece with no other tokens between it and the preceding one, it does not start a new comment:

```
i = 0     # This is a single comment over multiple lines.
  # The scanner merges these two pieces.
  # The comment continues here.
```

Documentation comments are comments that start with two ##. Documentation comments are tokens; they are only allowed at certain places in the input file as they belong to the syntax tree!

Multiline comments

Starting with version 0.13.0 of the language Nim supports multiline comments. They look like:

```
#[Comment here.
Multiple lines
are not a problem.]#
```

Multiline comments support nesting:

```
#[  #[ Multiline comment in already
   commented out code. ]#
proc p[T](x: T) = discard
]#
```

Multiline documentation comments also exist and support nesting too:

```
proc foo =
  ##[Long documentation comment
  here.
  ]##
```

Identifiers & Keywords

Identifiers in Nim can be any string of letters, digits and underscores, beginning with a letter. Two immediate following underscores `__` are not allowed:

```
letter ::= 'A'..'Z' | 'a'..'z' | '\x80'..'\xff'
digit ::= '0'..'9'
IDENTIFIER ::= letter ( ['_'] (letter | digit) )*
```

Currently any Unicode character with an ordinal value > 127 (non ASCII) is classified as a **letter** and may thus be part of an identifier but later versions of the language may assign some Unicode characters to belong to the operator characters instead.

The following keywords are reserved and cannot be used as identifiers:

```
addr and as asm atomic
bind block break
case cast concept const continue converter
defer discard distinct div do
elif else end enum except export
finally for from func
generic
if import in include interface is isnot iterator
let
macro method mixin mod
nil not notin
object of or out
proc ptr
raise ref return
shl shr static
template try tuple type
using
var
when while with without
xor
yield
```

Some keywords are unused; they are reserved for future developments of the language.

Identifier equality

Two identifiers are considered equal if the following algorithm returns true:

```
proc sameIdentifier(a, b: string): bool =
  a[0] == b[0] and
    a.replace("_", "").toLower == b.replace("_", "").toLower
```

That means only the first letters are compared in a case sensitive manner. Other letters are compared case insensitively and underscores are ignored.

This rather unorthodox way to do identifier comparisons is called partial case insensitivity and has some advantages over the conventional case sensitivity:

It allows programmers to mostly use their own preferred spelling style, be it humpStyle, snake_style or dash–style and libraries written by different programmers cannot use incompatible conventions. A Nim-aware editor or IDE can show the identifiers as preferred. Another advantage is that it frees the programmer from remembering the exact spelling of an identifier. The exception with respect to the first letter allows common code like `var foo: Foo` to be parsed unambiguously.

Historically, Nim was a fully style-insensitive language. This meant that it was not case-sensitive and underscores were ignored and there was no even a distinction between `foo` and `Foo`.

String literals

Terminal symbol in the grammar: `STR_LIT`.

String literals can be delimited by matching double quotes, and can contain the following escape sequences:

Escape sequence	Meaning
`\n`	newline
`\r`, `\c`	carriage return
`\l`	line feed
`\f`	form feed
`\t`	tabulator
`\v`	vertical tabulator
`\\`	backslash
`\"`	quotation mark
`\'`	apostrophe
`\` '0'..'9'+	character with decimal value d; all decimal digits directly following are used for the character
`\a`	alert
`\b`	backspace
`\e`	escape [ESC]
`\x` HH	character with hex value HH; exactly two hex digits are allowed

Strings in Nim may contain any 8-bit value, even embedded zeros. However some operations may interpret the first binary zero as a terminator.

Triple quoted string literals

Terminal symbol in the grammar: `TRIPLESTR_LIT`.

String literals can also be delimited by three double quotes `"""` ... `"""`. Literals in this form may run for several lines, may contain " and do not interpret any escape sequences. For convenience, when the opening `"""` is followed by a newline (there may be whitespace between the opening `"""` and the newline), the newline (and the preceding whitespace) is not included in the string. The ending of the string literal is defined by the pattern `"""[^"]`, so this:

```
""""long string within quotes""""
```

Produces:

```
"long string within quotes"
```

Raw string literals

Terminal symbol in the grammar: **`RSTR_LIT`**.

There are also raw string literals that are preceded with the letter `r` (or `R`) and are delimited by matching double quotes (just like ordinary string literals) and do not interpret the escape sequences. This is especially convenient for regular expressions or Windows paths:

```
var f = openFile(r"C:\texts\text.txt") # a raw string, so ``\t`` is no ta
```

To produce a single `"` within a raw string literal, it has to be doubled:

```
r"a""b"
```

Produces:

```
a"b
```

`r""""` is not possible with this notation, because the three leading quotes introduce a triple quoted string literal. `r"""` is the same as `"""` since triple quoted string literals do not interpret escape sequences either.

Generalized raw string literals

Terminal symbols in the grammar: **`GENERALIZED_STR_LIT`**, **`GENERALIZED_TRIPLESTR_LIT`**.

The construct **`identifier"string literal"`** (without whitespace between the identifier and the opening quotation mark) is a generalized raw string literal. It is a shortcut for the construct **`identifier(r"string literal")`**, so it denotes a procedure call with a raw string literal as its only argument. Generalized raw string literals are especially convenient for embedding mini languages directly into Nim (for example regular expressions).

The construct **`identifier"""string literal"""`** exists too. It is a shortcut for **`identifier("""string literal""")`**.

Character literals

Character literals are enclosed in single quotes `''` and can contain the same escape sequences as strings - with one exception: newline (`\n`) is not allowed as it may be wider than one character (often it is the pair CR/LF for example). Here are the valid escape sequences for character literals:

Escape sequence	Meaning
`\r`, `\c`	carriage return
`\l`	line feed

`\f`	form feed
`\t`	tabulator
`\v`	vertical tabulator
`\\`	backslash
`\"`	quotation mark
`\'`	apostrophe
`\` '0'..'9'+	character with decimal value d; all decimal digits directly following are used for the character
`\a`	alert
`\b`	backspace
`\e`	escape [ESC]
`\x` HH	character with hex value HH; exactly two hex digits are allowed

A character is not an Unicode character but a single byte. The reason for this is efficiency: for the overwhelming majority of use-cases, the resulting programs will still handle UTF-8 properly as UTF-8 was specially designed for this. Another reason is that Nim can thus support **`array[char, int]`** or **`set[char]`** efficiently as many algorithms rely on this feature. The *Rune* type is used for Unicode characters, it can represent any Unicode character. **`Rune`** is declared in the unicode module (unicode.html).

Numerical constants

Numerical constants are of a single type and have the form:

```
hexdigit = digit | 'A'..'F' | 'a'..'f'
octdigit = '0'..'7'
bindigit = '0'..'1'
HEX_LIT = '0' ('x' | 'X' ) hexdigit ( ['_'] hexdigit )*
DEC_LIT = digit ( ['_'] digit )*
OCT_LIT = '0' ('o' | 'c' | 'C') octdigit ( ['_'] octdigit )*
BIN_LIT = '0' ('b' | 'B' ) bindigit ( ['_'] bindigit )*

INT_LIT = HEX_LIT
        | DEC_LIT
        | OCT_LIT
        | BIN_LIT

INT8_LIT = INT_LIT ['\''] ('i' | 'I') '8'
INT16_LIT = INT_LIT ['\''] ('i' | 'I') '16'
INT32_LIT = INT_LIT ['\''] ('i' | 'I') '32'
INT64_LIT = INT_LIT ['\''] ('i' | 'I') '64'

UINT_LIT = INT_LIT ['\''] ('u' | 'U')
UINT8_LIT = INT_LIT ['\''] ('u' | 'U') '8'
UINT16_LIT = INT_LIT ['\''] ('u' | 'U') '16'
UINT32_LIT = INT_LIT ['\''] ('u' | 'U') '32'
UINT64_LIT = INT_LIT ['\''] ('u' | 'U') '64'

exponent = ('e' | 'E' ) ['+' | '-'] digit ( ['_'] digit )*
FLOAT_LIT = digit (['_'] digit)* (('.' (['_'] digit)* [exponent]) |expone
FLOAT32_SUFFIX = ('f' | 'F') ['32']
FLOAT32_LIT = HEX_LIT '\'' FLOAT32_SUFFIX
            | (FLOAT_LIT | DEC_LIT | OCT_LIT | BIN_LIT) ['\''] FLOAT32_SU
FLOAT64_SUFFIX = ( ('f' | 'F') '64' ) | 'd' | 'D'
FLOAT64_LIT = HEX_LIT '\'' FLOAT64_SUFFIX
            | (FLOAT_LIT | DEC_LIT | OCT_LIT | BIN_LIT) ['\''] FLOAT64_SU
```

As can be seen in the productions, numerical constants can contain underscores for readability.

Integer and floating point literals may be given in decimal (no prefix), binary (prefix `0b`), octal (prefix `0o` or `0c`) and hexadecimal (prefix `0x`) notation.

There exists a literal for each numerical type that is defined. The suffix starting with an apostrophe (''') is called a type suffix. Literals without a type suffix are of the type `int`, unless the literal contains a dot or `E|e` in which case it is of type `float`. For notational convenience the apostrophe of a type suffix is optional if it is not ambiguous (only hexadecimal floating point literals with a type suffix can be ambiguous).

The type suffixes are:

Type Suffix	**Resulting type of literal**
`'i8`	int8
`'i16`	int16
`'i32`	int32
`'i64`	int64
`'u`	uint
`'u8`	uint8
`'u16`	uint16
`'u32`	uint32
`'u64`	uint64
`'f`	float32
`'d`	float64
`'f32`	float32
`'f64`	float64
`'f128`	float128

Floating point literals may also be in binary, octal or hexadecimal notation: `0B0_10001110100_0000101001000111101011101111111011000101001101001001'f64` is approximately 1.72826e35 according to the IEEE floating point standard.

Literals are bounds checked so that they fit the datatype. Non base-10 literals are used mainly for flags and bit pattern representations, therefore bounds checking is done on bit width, not value range. If the literal fits in the bit width of the datatype, it is accepted. Hence: 0b10000000'u8 == 0x80'u8 == 128, but, 0b10000000'i8 == 0x80'i8 == -1 instead of causing an overflow error.

Operators

Nim allows user defined operators. An operator is any combination of the following characters:

```
=     +     -     *     /     <     >
@     $     ~     &     %     |
!     ?     ^     .     :     \
```

These keywords are also operators:
`and or not xor shl shr div mod in notin is isnot of`.

=, :, :: are not available as general operators; they are used for other notational purposes.

`*:` is as a special case treated as the two tokens * and : (to support `var v*: T`).

Other tokens

The following strings denote other tokens:

```
`   (    )    {    }    [    ]    ,  ;   [.   .]  {.   .}  (.  .)
```

The slice operator .. takes precedence over other tokens that contain a dot: {..} are the three tokens {, .., } and not the two tokens {., .}.

Syntax

This section lists Nim's standard syntax. How the parser handles the indentation is already described in the Lexical Analysis section.

Nim allows user-definable operators. Binary operators have 11 different levels of precedence.

Associativity

Binary operators whose first character is `^` are right-associative, all other binary operators are left-associative.

```
proc `^/`(x, y: float): float =
  # a right-associative division operator
  result = x / y
echo 12 ^/ 4 ^/ 8 # 24.0 (4 / 8 = 0.5, then 12 / 0.5 = 24.0)
echo 12  / 4  / 8 # 0.375 (12 / 4 = 3.0, then 3 / 8 = 0.375)
```

Precedence

Unary operators always bind stronger than any binary operator: `$a + b` is `($a) + b` and not `$(a + b)`.

If an unary operator's first character is `@` it is a sigil-like operator which binds stronger than a `primarySuffix`: `@x.abc` is parsed as `(@x).abc` whereas `$x.abc` is parsed as `$(x.abc)`.

For binary operators that are not keywords the precedence is determined by the following rules:

Operators ending in either `->`, `~>` or `=>` are called arrow like, and have the lowest precedence of all operators.

If the operator ends with `=` and its first character is none of `<`, `>`, `!`, `=`, `~`, `?`, it is an *assignment operator* which has the second lowest precedence.

Otherwise precedence is determined by the first character.

Precedence level	**Operators**	**First character**	**Terminal symbol**
10 (highest)		`$ ^`	OP10
9	`* / div mod shl shr %`	`* % \ /`	OP9
8	`+ -`	`+ - ~ |`	OP8
7	`&`	`&`	OP7
6	`..`	`.`	OP6
5	`== <= < >= > != in notin is isnot not of`	`= < > !`	OP5
4	`and`		OP4
3	`or xor`		OP3
2		`@ : ?`	OP2
1	*assignment operator* (like `+=`, `*=`)		OP1
0 (lowest)	*arrow like operator* (like `->`, `=>`)		OP0

Whether an operator is used a prefix operator is also affected by preceding whitespace (this parsing change was introduced with version 0.13.0):

```
echo $foo
# is parsed as
echo($foo)
```

Grammar

The grammar's start symbol is **`module`**.

```
module = stmt ^* (';' / IND{=})
comma = ',' COMMENT?
semicolon = ';' COMMENT?
colon = ':' COMMENT?
colcom = ':' COMMENT?
operator =  OP0 | OP1 | OP2 | OP3 | OP4 | OP5 | OP6 | OP7 | OP8 | OP9
         | 'or' | 'xor' | 'and'
         | 'is' | 'isnot' | 'in' | 'notin' | 'of'
         | 'div' | 'mod' | 'shl' | 'shr' | 'not' | 'static' | '..'
prefixOperator = operator
optInd = COMMENT?
optPar = (IND{>} | IND{=})?
simpleExpr = arrowExpr (OP0 optInd arrowExpr)* pragma?
arrowExpr = assignExpr (OP1 optInd assignExpr)*
assignExpr = orExpr (OP2 optInd orExpr)*
orExpr = andExpr (OP3 optInd andExpr)*
andExpr = cmpExpr (OP4 optInd cmpExpr)*
cmpExpr = sliceExpr (OP5 optInd sliceExpr)*
sliceExpr = ampExpr (OP6 optInd ampExpr)*
ampExpr = plusExpr (OP7 optInd plusExpr)*
plusExpr = mulExpr (OP8 optInd mulExpr)*
mulExpr = dollarExpr (OP9 optInd dollarExpr)*
dollarExpr = primary (OP10 optInd primary)*
symbol = '`' (KEYW|IDENT|literal|(operator|'('|')'|'['|']'|'{'|'}'|'=')+)+ '`'
       | IDENT | KEYW
exprColonEqExpr = expr (':'|'=' expr)?
exprList = expr ^+ comma
dotExpr = expr '.' optInd symbol
qualifiedIdent = symbol ('.' optInd symbol)?
exprColonEqExprList = exprColonEqExpr (comma exprColonEqExpr)* (comma)?
setOrTableConstr = '{' ((exprColonEqExpr comma)* | ':' ) '}'
castExpr = 'cast' '[' optInd typeDesc optPar ']' '(' optInd expr optPar ')'
parKeyw = 'discard' | 'include' | 'if' | 'while' | 'case' | 'try'
        | 'finally' | 'except' | 'for' | 'block' | 'const' | 'let'
        | 'when' | 'var' | 'mixin'
par = '(' optInd
          ( &parKeyw complexOrSimpleStmt ^+ ';'
          | ';' complexOrSimpleStmt ^+ ';'
          | pragmaStmt
          | simpleExpr ( ('=' expr (';' complexOrSimpleStmt ^+ ';' )? )
                       | (':' expr (',' exprColonEqExpr     ^+ ',' )? ) ) )
          optPar ')'
literal = | INT_LIT | INT8_LIT | INT16_LIT | INT32_LIT | INT64_LIT
          | UINT_LIT | UINT8_LIT | UINT16_LIT | UINT32_LIT | UINT64_LIT
          | FLOAT_LIT | FLOAT32_LIT | FLOAT64_LIT
          | STR_LIT | RSTR_LIT | TRIPLESTR_LIT
          | CHAR_LIT
          | NIL
generalizedLit = GENERALIZED_STR_LIT | GENERALIZED_TRIPLESTR_LIT
identOrLiteral = generalizedLit | symbol | literal
               | par | arrayConstr | setOrTableConstr
               | castExpr
tupleConstr = '(' optInd (exprColonEqExpr comma?)* optPar ')'
arrayConstr = '[' optInd (exprColonEqExpr comma?)* optPar ']'
primarySuffix = '(' (exprColonEqExpr comma?)* ')' doBlocks?
      | doBlocks
      | '.' optInd symbol generalizedLit?
      | '[' optInd indexExprList optPar ']'
      | '{' optInd indexExprList optPar '}'
```

Types

All expressions have a type which is known at compile time. Nim is statically typed. One can declare new types, which is in essence defining an identifier that can be used to denote this custom type.

These are the major type classes:

- ordinal types (consist of integer, bool, character, enumeration (and subranges thereof) types)
- floating point types
- string type
- structured types
- reference (pointer) type
- procedural type
- generic type

Ordinal types

Ordinal types have the following characteristics:

- Ordinal types are countable and ordered. This property allows the operation of functions as **`inc`**, **`ord`**, **`dec`** on ordinal types to be defined.
- Ordinal values have a smallest possible value. Trying to count further down than the smallest value gives a checked runtime or static error.
- Ordinal values have a largest possible value. Trying to count further than the largest value gives a checked runtime or static error.

Integers, bool, characters and enumeration types (and subranges of these types) belong to ordinal types. For reasons of simplicity of implementation the types **`uint`** and **`uint64`** are not ordinal types.

Pre-defined integer types

These integer types are pre-defined:

`int`
: the generic signed integer type; its size is platform dependent and has the same size as a pointer. This type should be used in general. An integer literal that has no type suffix is of this type.

intXX
: additional signed integer types of XX bits use this naming scheme (example: int16 is a 16 bit wide integer). The current implementation supports **`int8`**, **`int16`**, **`int32`**, **`int64`**. Literals of these types have the suffix 'iXX.

`uint`
: the generic unsigned integer type; its size is platform dependent and has the same size as a pointer. An integer literal with the type suffix **`'u`** is of this type.

uintXX
: additional signed integer types of XX bits use this naming scheme (example: uint16 is a 16 bit wide unsigned integer). The current implementation supports **`uint8`**, **`uint16`**, **`uint32`**, **`uint64`**. Literals of these types have the suffix 'uXX. Unsigned operations all wrap around; they cannot lead to over- or underflow errors.

In addition to the usual arithmetic operators for signed and unsigned integers (**`+ - *`** etc.) there are also operators that formally work on *signed* integers but treat their arguments as *unsigned*: They are mostly provided for backwards compatibility with older versions of the language that lacked unsigned integer types. These unsigned operations for signed integers use the `%` suffix as convention:

operation	meaning
`a +% b`	unsigned integer addition
`a -% b`	unsigned integer subtraction
`a *% b`	unsigned integer multiplication
`a /% b`	unsigned integer division
`a %% b`	unsigned integer modulo operation
`a <% b`	treat **a** and **b** as unsigned and compare
`a <=% b`	treat **a** and **b** as unsigned and compare
`ze(a)`	extends the bits of **a** with zeros until it has the width of the `int` type
`toU8(a)`	treats **a** as unsigned and converts it to an unsigned integer of 8 bits (but still the `int8` type)
`toU16(a)`	treats **a** as unsigned and converts it to an unsigned integer of 16 bits (but still the `int16` type)
`toU32(a)`	treats **a** as unsigned and converts it to an unsigned integer of 32 bits (but still the `int32` type)

Automatic type conversion is performed in expressions where different kinds of integer types are used: the smaller type is converted to the larger.

A narrowing type conversion converts a larger to a smaller type (for example `int32 -> int16`. A widening type conversion converts a smaller type to a larger type (for example `int16 -> int32`). In Nim only widening type conversions are *implicit*:

```
var myInt16 = 5i16
var myInt: int
myInt16 + 34     # of type ``int16``
myInt16 + myInt  # of type ``int``
myInt16 + 2i32   # of type ``int32``
```

However, `int` literals are implicitly convertible to a smaller integer type if the literal's value fits this smaller type and such a conversion is less expensive than other implicit conversions, so `myInt16 + 34` produces an `int16` result.

For further details, see Convertible relation.

Subrange types

A subrange type is a range of values from an ordinal type (the base type). To define a subrange type, one must specify it's limiting values: the lowest and highest value of the type:

```
type
  Subrange = range[0..5]
```

`Subrange` is a subrange of an integer which can only hold the values 0 to 5. Assigning any other value to a variable of type `Subrange` is a checked runtime error (or static error if it can be statically determined). Assignments from the base type to one of its subrange types (and vice versa) are allowed.

A subrange type has the same size as its base type (`int` in the example).

Nim requires interval arithmetic for subrange types over a set of built-in operators that involve constants: `x %% 3` is of type `range[0..2]`. The following built-in operators for integers are affected by this rule: `-`, `+`, `*`, `min`, `max`, `succ`, `pred`, `mod`, `div`, `%%`, `and` (bitwise `and`).

Bitwise `and` only produces a `range` if one of its operands is a constant *x* so that (x+1) is a power of two. (Bitwise `and` is then a `%%` operation.)

This means that the following code is accepted:

```
case (x and 3) + 7
of 7: echo "A"
of 8: echo "B"
of 9: echo "C"
of 10: echo "D"
# note: no ``else`` required as (x and 3) + 7 has the type: range[7..10]
```

Pre-defined floating point types

The following floating point types are pre-defined:

`float`
the generic floating point type; its size is platform dependent (the compiler chooses the processor's fastest floating point type). This type should be used in general.

floatXX
an implementation may define additional floating point types of XX bits using this naming scheme (example: float64 is a 64 bit wide float). The current implementation supports **`float32`** and **`float64`**. Literals of these types have the suffix 'fXX.

Automatic type conversion in expressions with different kinds of floating point types is performed: See Convertible relation for further details. Arithmetic performed on floating point types follows the IEEE standard. Integer types are not converted to floating point types automatically and vice versa.

The IEEE standard defines five types of floating-point exceptions:

- Invalid: operations with mathematically invalid operands, for example 0.0/0.0, sqrt(-1.0), and log(-37.8).
- Division by zero: divisor is zero and dividend is a finite nonzero number, for example 1.0/0.0.
- Overflow: operation produces a result that exceeds the range of the exponent, for example MAXDOUBLE+0.0000000000001e308.
- Underflow: operation produces a result that is too small to be represented as a normal number, for example, MINDOUBLE * MINDOUBLE.
- Inexact: operation produces a result that cannot be represented with infinite precision, for example, 2.0 / 3.0, log(1.1) and 0.1 in input.

The IEEE exceptions are either ignored at runtime or mapped to the Nim exceptions: FloatInvalidOpError, FloatDivByZeroError, FloatOverflowError, FloatUnderflowError, and FloatInexactError. These exceptions inherit from the FloatingPointError base class.

Nim provides the pragmas NaNChecks and InfChecks to control whether the IEEE exceptions are ignored or trap a Nim exception:

```
{.NanChecks: on, InfChecks: on.}
var a = 1.0
var b = 0.0
echo b / b # raises FloatInvalidOpError
echo a / b # raises FloatOverflowError
```

In the current implementation **`FloatDivByZeroError`** and **`FloatInexactError`** are never raised. **`FloatOverflowError`** is raised instead of **`FloatDivByZeroError`**. There is also a floatChecks pragma that is a short-cut for the combination of **`NaNChecks`** and **`InfChecks`** pragmas. **`floatChecks`** are turned off as default.

The only operations that are affected by the **`floatChecks`** pragma are the **`+`**, **`-`**, **`*`**, **`/`** operators for floating point types.

An implementation should always use the maximum precision available to evaluate floating pointer values at compile time; this means expressions like `0.09'f32 + 0.01'f32 == 0.09'f64 + 0.01'f64` are true.

Boolean type

The boolean type is named bool in Nim and can be one of the two pre-defined values **`true`** and **`false`**. Conditions in **`while`**, **`if`**, **`elif`**, **`when`**-statements need to be of type **`bool`**.

This condition holds:

```
ord(false) == 0 and ord(true) == 1
```

The operators **`not, and, or, xor, <, <=, >, >=, !=, ==`** are defined for the bool type. The **`and`** and **`or`** operators perform short-cut evaluation. Example:

```
while p != nil and p.name != "xyz":
  # p.name is not evaluated if p == nil
  p = p.next
```

The size of the bool type is one byte.

Character type

The character type is named **`char`** in Nim. Its size is one byte. Thus it cannot represent an UTF-8 character, but a part of it. The reason for this is efficiency: for the overwhelming majority of use-cases, the resulting programs will still handle UTF-8 properly as UTF-8 was specially designed for this. Another reason is that Nim can support **`array[char, int]`** or **`set[char]`** efficiently as many algorithms rely on this feature. The *Rune* type is used for Unicode characters, it can represent any Unicode character. **`Rune`** is declared in the unicode module (unicode.html).

Enumeration types

Enumeration types define a new type whose values consist of the ones specified. The values are ordered. Example:

```
type
  Direction = enum
    north, east, south, west
```

Now the following holds:

```
ord(north) == 0
ord(east) == 1
ord(south) == 2
ord(west) == 3
```

Thus, north < east < south < west. The comparison operators can be used with enumeration types.

For better interfacing to other programming languages, the fields of enum types can be assigned an explicit ordinal value. However, the ordinal values have to be in ascending order. A field whose ordinal value is not explicitly given is assigned the value of the previous field `+ 1`.

An explicit ordered enum can have *holes*:

```
type
  TokenType = enum
    a = 2, b = 4, c = 89 # holes are valid
```

However, it is then not an ordinal anymore, so it is not possible to use these enums as an index type for arrays. The procedures **inc**, **dec**, **succ** and **pred** are not available for them either.

The compiler supports the built-in stringify operator **$** for enumerations. The stringify's result can be controlled by explicitly giving the string values to use:

```
type
  MyEnum = enum
    valueA = (0, "my value A"),
    valueB = "value B",
    valueC = 2,
    valueD = (3, "abc")
```

As can be seen from the example, it is possible to both specify a field's ordinal value and its string value by using a tuple. It is also possible to only specify one of them.

An enum can be marked with the **pure** pragma so that it's fields are not added to the current scope, so they always need to be accessed via **MyEnum.value**:

```
type
  MyEnum {.pure.} = enum
    valueA, valueB, valueC, valueD

echo valueA # error: Unknown identifier
echo MyEnum.valueA # works
```

String type

All string literals are of the type **string**. A string in Nim is very similar to a sequence of characters. However, strings in Nim are both zero-terminated and have a length field. One can retrieve the length with the builtin **len** procedure; the length never counts the terminating zero. The assignment operator for strings always copies the string. The **&** operator concatenates strings.

Most native Nim types support conversion to strings with the special **$** proc. When calling the **echo** proc, for example, the built-in stringify operation for the parameter is called:

```
echo 3 # calls `$` for `int`
```

Whenever a user creates a specialized object, implementation of this procedure provides for **string** representation.

```
type
  Person = object
    name: string
    age: int

proc `$`(p: Person): string = # `$` always returns a string
  result = p.name & " is " &
          $p.age & # we *need* the `$` in front of p.age, which
                   # is natively an integer, to convert it to
                   # a string
          " years old."
```

While **$p.name** can also be used, the **$** operation on a string does nothing. Note that we cannot rely on automatic conversion from an **int** to a **string** like we can for the **echo** proc.

Strings are compared by their lexicographical order. All comparison operators are available. Strings can be indexed like arrays (lower bound is 0). Unlike arrays, they can be used in case statements:

```
case paramStr(i)
of "-v": incl(options, optVerbose)
of "-h", "-?": incl(options, optHelp)
else: write(stdout, "invalid command line option!\n")
```

Per convention, all strings are UTF-8 strings, but this is not enforced. For example, when reading strings from binary files, they are merely a sequence of bytes. The index operation **s[i]** means the i-th *char* of **s**, not the i-th *unichar*. The iterator **runes** from the unicode module (unicode.html) can be used for iteration over all Unicode characters.

cstring type

The **cstring** type meaning *compatible string* is the native representation of a string for the compilation backend. For the C backend the **cstring** type represents a pointer to a zero-terminated char array compatible to the type **char*** in Ansi C. Its primary purpose lies in easy interfacing with C. The index operation **s[i]** means the i-th *char* of **s**; however no bounds checking for **cstring** is performed making the index operation unsafe.

A Nim **string** is implicitly convertible to **cstring** for convenience. If a Nim string is passed to a C-style variadic proc, it is implicitly converted to **cstring** too:

```
proc printf(formatstr: cstring) {.importc: "printf", varargs,
                                  header: "<stdio.h>".}

printf("This works %s", "as expected")
```

Even though the conversion is implicit, it is not *safe*: The garbage collector does not consider a **cstring** to be a root and may collect the underlying memory. However in practice this almost never happens as the GC considers stack roots conservatively. One can use the builtin procs **GC_ref** and **GC_unref** to keep the string data alive for the rare cases where it does not work.

A *$* proc is defined for cstrings that returns a string. Thus to get a nim string from a cstring:

```
var str: string = "Hello!"
var cstr: cstring = str
var newstr: string = $cstr
```

Structured types

A variable of a structured type can hold multiple values at the same time. Structured types can be nested to unlimited levels. Arrays, sequences, tuples, objects and sets belong to the structured types.

Array and sequence types

Arrays are a homogeneous type, meaning that each element in the array has the same type. Arrays always have a fixed length which is specified at compile time (except for open arrays). They can be indexed by any ordinal type. A parameter **A** may be an *open array*, in which case it is indexed by integers from 0 to **len(A)-1**. An array expression may be constructed by the array constructor **[]**. The element type of this array expression is inferred from the type of the first element. All other elements need to be implicitly convertable to this type.

Sequences are similar to arrays but of dynamic length which may change during runtime (like strings). Sequences are implemented as growable arrays, allocating pieces of memory as items are added. A sequence **`S`** is always indexed by integers from 0 to **`len(S)-1`** and its bounds are checked. Sequences can be constructed by the array constructor `[]` in conjunction with the array to sequence operator **`@`**. Another way to allocate space for a sequence is to call the built-in **`newSeq`** procedure.

A sequence may be passed to a parameter that is of type *open array*.

Example:

```
type
  IntArray = array[0..5, int] # an array that is indexed with 0..5
  IntSeq = seq[int] # a sequence of integers
var
  x: IntArray
  y: IntSeq
x = [1, 2, 3, 4, 5, 6]  # [] is the array constructor
y = @[1, 2, 3, 4, 5, 6] # the @ turns the array into a sequence

let z = [1.0, 2, 3, 4] # the type of z is array[0..3, float]
```

The lower bound of an array or sequence may be received by the built-in proc **`low()`**, the higher bound by **`high()`**. The length may be received by **`len()`**. **`low()`** for a sequence or an open array always returns 0, as this is the first valid index. One can append elements to a sequence with the **`add()`** proc or the **`&`** operator, and remove (and get) the last element of a sequence with the **`pop()`** proc.

The notation **`x[i]`** can be used to access the i-th element of **`x`**.

Arrays are always bounds checked (at compile-time or at runtime). These checks can be disabled via pragmas or invoking the compiler with the **`--boundChecks:off`** command line switch.

Open arrays

Often fixed size arrays turn out to be too inflexible; procedures should be able to deal with arrays of different sizes. The openarray type allows this; it can only be used for parameters. Openarrays are always indexed with an **`int`** starting at position 0. The **`len`**, **`low`** and **`high`** operations are available for open arrays too. Any array with a compatible base type can be passed to an openarray parameter, the index type does not matter. In addition to arrays sequences can also be passed to an open array parameter.

The openarray type cannot be nested: multidimensional openarrays are not supported because this is seldom needed and cannot be done efficiently.

```
proc testOpenArray(x: openArray[int]) = echo repr(x)

testOpenArray([1,2,3])  # array[]
testOpenArray(@[1,2,3]) # seq[]
```

Varargs

A **`varargs`** parameter is an openarray parameter that additionally allows to pass a variable number of arguments to a procedure. The compiler converts the list of arguments to an array implicitly:

```
proc myWriteln(f: File, a: varargs[string]) =
  for s in items(a):
    write(f, s)
  write(f, "\n")

myWriteln(stdout, "abc", "def", "xyz")
# is transformed to:
myWriteln(stdout, ["abc", "def", "xyz"])
```

This transformation is only done if the varargs parameter is the last parameter in the procedure header. It is also possible to perform type conversions in this context:

```
proc myWriteln(f: File, a: varargs[string, `$`]) =
  for s in items(a):
    write(f, s)
  write(f, "\n")

myWriteln(stdout, 123, "abc", 4.0)
# is transformed to:
myWriteln(stdout, [$123, $"def", $4.0])
```

In this example `$` is applied to any argument that is passed to the parameter `a`. (Note that `$` applied to strings is a nop.)

Note that an explicit array constructor passed to a `varargs` parameter is not wrapped in another implicit array construction:

```
proc takeV[T](a: varargs[T]) = discard

takeV([123, 2, 1]) # takeV's T is "int", not "array of int"
```

`varargs[typed]` is treated specially: It matches a variable list of arguments of arbitrary type but *always* constructs an implicit array. This is required so that the builtin `echo` proc does what is expected:

```
proc echo*(x: varargs[typed, `$`]) {...}

echo @[1, 2, 3]
# prints "@[1, 2, 3]" and not "123"
```

Tuples and object types

A variable of a tuple or object type is a heterogeneous storage container. A tuple or object defines various named *fields* of a type. A tuple also defines an *order* of the fields. Tuples are meant for heterogeneous storage types with no overhead and few abstraction possibilities. The constructor `()` can be used to construct tuples. The order of the fields in the constructor must match the order of the tuple's definition. Different tuple-types are *equivalent* if they specify the same fields of the same type in the same order. The *names* of the fields also have to be identical.

The assignment operator for tuples copies each component. The default assignment operator for objects copies each component. Overloading of the assignment operator for objects is not possible, but this will change in future versions of the compiler.

```
type
  Person = tuple[name: string, age: int] # type representing a person:
                                         # a person consists of a name
                                         # and an age
var
  person: Person
person = (name: "Peter", age: 30)
# the same, but less readable:
person = ("Peter", 30)
```

The implementation aligns the fields for best access performance. The alignment is compatible with the way the C compiler does it.

For consistency with **`object`** declarations, tuples in a **`type`** section can also be defined with indentation instead of `[]`:

```
type
  Person = tuple   # type representing a person
    name: string   # a person consists of a name
    age: natural   # and an age
```

Objects provide many features that tuples do not. Object provide inheritance and information hiding. Objects have access to their type at runtime, so that the **`of`** operator can be used to determine the object's type. The **`of`** operator is similar to the **`instanceof`** operator in Java.

```
type
  Person = object of RootObj
    name*: string   # the * means that `name` is accessible from other modules
    age: int        # no * means that the field is hidden

  Student = ref object of Person # a student is a person
    id: int                      # with an id field

var
  student: Student
  person: Person
assert(student of Student) # is true
assert(student of Person) # also true
```

Object fields that should be visible from outside the defining module, have to be marked by `*`. In contrast to tuples, different object types are never *equivalent*. Objects that have no ancestor are implicitly **`final`** and thus have no hidden type field. One can use the **`inheritable`** pragma to introduce new object roots apart from **`system.RootObj`**.

Object construction

Objects can also be created with an object construction expression that has the syntax **`T(fieldA: valueA, fieldB: valueB, ...)`** where **`T`** is an **`object`** type or a **`ref object`** type:

```
var student = Student(name: "Anton", age: 5, id: 3)
```

Note that, unlike tuples, objects require the field names along with their values. For a **`ref object`** type **`system.new`** is invoked implicitly.

Object variants

Often an object hierarchy is overkill in certain situations where simple variant types are needed.

An example:

```
# This is an example how an abstract syntax tree could be modelled in Nim
type
  NodeKind = enum  # the different node types
    nkInt,          # a leaf with an integer value
    nkFloat,        # a leaf with a float value
    nkString,       # a leaf with a string value
    nkAdd,          # an addition
    nkSub,          # a subtraction
    nkIf            # an if statement
  Node = ref NodeObj
  NodeObj = object
    case kind: NodeKind  # the ``kind`` field is the discriminator
    of nkInt: intVal: int
    of nkFloat: floatVal: float
    of nkString: strVal: string
    of nkAdd, nkSub:
      leftOp, rightOp: Node
    of nkIf:
      condition, thenPart, elsePart: Node

# create a new case object:
var n = Node(kind: nkIf, condition: nil)
# accessing n.thenPart is valid because the ``nkIf`` branch is active:
n.thenPart = Node(kind: nkFloat, floatVal: 2.0)

# the following statement raises an `FieldError` exception, because
# n.kind's value does not fit and the ``nkString`` branch is not active:
n.strVal = ""

# invalid: would change the active object branch:
n.kind = nkInt

var x = Node(kind: nkAdd, leftOp: Node(kind: nkInt, intVal: 4),
                          rightOp: Node(kind: nkInt, intVal: 2))
# valid: does not change the active object branch:
x.kind = nkSub
```

As can been seen from the example, an advantage to an object hierarchy is that no casting between different object types is needed. Yet, access to invalid object fields raises an exception.

The syntax of **`case`** in an object declaration follows closely the syntax of the **`case`** statement: The branches in a **`case`** section may be indented too.

In the example the **`kind`** field is called the discriminator: For safety its address cannot be taken and assignments to it are restricted: The new value must not lead to a change of the active object branch. For an object branch switch **`system.reset`** has to be used. Also, when the fields of a particular branch are specified during object construction, the correct value for the discriminator must be supplied at compile-time.

Set type

The set type models the mathematical notion of a set. The set's basetype can

only be an ordinal type of a certain size, namely:

- **`int8-int16`**

- `uint8/byte-uint16`
- `char`
- `enum`

or equivalent. The reason is that sets are implemented as high performance bit vectors. Attempting to declare a set with a larger type will result in an error:

```
var s: set[int64] # Error: set is too large
```

Sets can be constructed via the set constructor: `{}` is the empty set. The empty set is type compatible with any concrete set type. The constructor can also be used to include elements (and ranges of elements):

```
type
  CharSet = set[char]
var
  x: CharSet
x = {'a'..'z', '0'..'9'} # This constructs a set that contains the
                         # letters from 'a' to 'z' and the digits
                         # from '0' to '9'
```

These operations are supported by sets:

operation	meaning
`A + B`	union of two sets
`A * B`	intersection of two sets
`A - B`	difference of two sets (A without B's elements)
`A == B`	set equality
`A <= B`	subset relation (A is subset of B or equal to B)
`A < B`	strong subset relation (A is a real subset of B)
`e in A`	set membership (A contains element e)
`e notin A`	A does not contain element e
`contains(A, e)`	A contains element e
`card(A)`	the cardinality of A (number of elements in A)
`incl(A, elem)`	same as `A = A + {elem}`
`excl(A, elem)`	same as `A = A - {elem}`

Sets are often used to define a type for the *flags* of a procedure. This is a much cleaner (and type safe) solution than just defining integer constants that should be `or`'ed together.

Reference and pointer types

References (similar to pointers in other programming languages) are a way to introduce many-to-one relationships. This means different references can point to and modify the same location in memory (also called aliasing).

Nim distinguishes between traced and untraced references. Untraced references are also called *pointers*. Traced references point to objects of a garbage collected heap, untraced references point to manually allocated objects or to objects somewhere else in memory. Thus untraced references are *unsafe*. However for certain low-level operations (accessing the hardware) untraced references are unavoidable.

Traced references are declared with the **ref** keyword, untraced references are declared with the **ptr** keyword. In general, a *ptr T* is implicitly convertible to the *pointer* type.

An empty subscript `[]` notation can be used to derefer a reference, the `addr` procedure returns the address of an item. An address is always an untraced reference. Thus the usage of `addr` is an *unsafe* feature.

The `.` (access a tuple/object field operator) and `[]` (array/string/sequence index operator) operators perform implicit dereferencing operations for reference types:

```
type
  Node = ref NodeObj
  NodeObj = object
    le, ri: Node
    data: int

var
  n: Node
new(n)
n.data = 9
# no need to write n[].data; in fact n[].data is highly discouraged!
```

Automatic dereferencing is also performed for the first argument of a routine call. But currently this feature has to be only enabled via **`{.experimental.}`**:

```
{.experimental.}

proc depth(x: NodeObj): int = ...

var
  n: Node
new(n)
echo n.depth
# no need to write n[].depth either
```

In order to simplify structural type checking, recursive tuples are not valid:

```
# invalid recursion
type MyTuple = tuple[a: ref MyTuple]
```

Likewise **`T = ref T`** is an invalid type.

As a syntactical extension **`object`** types can be anonymous if declared in a type section via the **`ref object`** or **`ptr object`** notations. This feature is useful if an object should only gain reference semantics:

```
type
  Node = ref object
    le, ri: Node
    data: int
```

To allocate a new traced object, the built-in procedure **`new`** has to be used. To deal with untraced memory, the procedures **`alloc`**, **`dealloc`** and **`realloc`** can be used. The documentation of the system module contains further information.

If a reference points to *nothing*, it has the value **`nil`**.

Special care has to be taken if an untraced object contains traced objects like traced references, strings or sequences: in order to free everything properly, the built-in procedure **`GCunref`** has to be called before freeing the untraced memory manually:

```
type
  Data = tuple[x, y: int, s: string]

# allocate memory for Data on the heap:
var d = cast[ptr Data](alloc0(sizeof(Data)))

# create a new string on the garbage collected heap:
d.s = "abc"

# tell the GC that the string is not needed anymore:
GCunref(d.s)

# free the memory:
dealloc(d)
```

Without the **`GCunref`** call the memory allocated for the **`d.s`** string would never be freed. The example also demonstrates two important features for low level programming: the **`sizeof`** proc returns the size of a type or value in bytes. The **`cast`** operator can circumvent the type system: the compiler is forced to treat the result of the **`alloc0`** call (which returns an untyped pointer) as if it would have the type **`ptr Data`**. Casting should only be done if it is unavoidable: it breaks type safety and bugs can lead to mysterious crashes.

Note: The example only works because the memory is initialized to zero (**`alloc0`** instead of **`alloc`** does this): **`d.s`** is thus initialized to **`nil`** which the string assignment can handle. One needs to know low level details like this when mixing garbage collected data with unmanaged memory.

Not nil annotation

All types for that **`nil`** is a valid value can be annotated to exclude **`nil`** as a valid value with the **`not nil`** annotation:

```
type
  PObject = ref TObj not nil
  TProc = (proc (x, y: int)) not nil

proc p(x: PObject) =
  echo "not nil"

# compiler catches this:
p(nil)

# and also this:
var x: PObject
p(x)
```

The compiler ensures that every code path initializes variables which contain non nilable pointers. The details of this analysis are still to be specified here.

Memory regions

The types **`ref`** and **`ptr`** can get an optional **`region`** annotation. A region has to be an object type.

Regions are very useful to separate user space and kernel memory in the development of OS kernels:

```
type
  Kernel = object
  Userspace = object

var a: Kernel ptr Stat
var b: Userspace ptr Stat

# the following does not compile as the pointer types are incompatible:
a = b
```

As the example shows **ptr** can also be used as a binary operator, **region ptr T** is a shortcut for **ptr[region, T]**.

In order to make generic code easier to write **ptr T** is a subtype of **ptr[R, T]** for any **R**.

Furthermore the subtype relation of the region object types is lifted to the pointer types: If **A <: B** then **ptr[A, T] <: ptr[B, T]**. This can be used to model subregions of memory. As a special typing rule **ptr[R, T]** is not compatible to **pointer** to prevent the following from compiling:

```
# from system
proc dealloc(p: pointer)

# wrap some scripting language
type
  PythonsHeap = object
  PyObjectHeader = object
    rc: int
    typ: pointer
  PyObject = ptr[PythonsHeap, PyObjectHeader]

proc createPyObject(): PyObject {.importc: "...".}
proc destroyPyObject(x: PyObject) {.importc: "...".}

var foo = createPyObject()
# type error here, how convenient:
dealloc(foo)
```

Future directions:

- Memory regions might become available for **string** and **seq** too.
- Builtin regions like **private**, **global** and **local** will prove very useful for the upcoming OpenCL target.
- Builtin "regions" can model **lent** and **unique** pointers.
- An assignment operator can be attached to a region so that proper write barriers can be generated. This would imply that the GC can be implemented completely in user-space.

Procedural type

A procedural type is internally a pointer to a procedure. **nil** is an allowed value for variables of a procedural type. Nim uses procedural types to achieve functional programming techniques.

Examples:

```
proc printItem(x: int) = ...

proc forEach(c: proc (x: int) {.cdecl.}) =
  ...

forEach(printItem)  # this will NOT compile because calling conventions differ
```

```
type
  OnMouseMove = proc (x, y: int) {.closure.}

proc onMouseMove(mouseX, mouseY: int) =
  # has default calling convention
  echo "x: ", mouseX, " y: ", mouseY

proc setOnMouseMove(mouseMoveEvent: OnMouseMove) = discard

# ok, 'onMouseMove' has the default calling convention, which is compatible
# to 'closure':
setOnMouseMove(onMouseMove)
```

A subtle issue with procedural types is that the calling convention of the procedure influences the type compatibility: procedural types are only compatible if they have the same calling convention. As a special extension, a procedure of the calling convention `nimcall` can be passed to a parameter that expects a proc of the calling convention `closure`.

Nim supports these calling conventions:

nimcall
is the default convention used for a Nim **proc**. It is the same as `fastcall`, but only for C compilers that support `fastcall`.

closure
is the default calling convention for a **procedural type** that lacks any pragma annotations. It indicates that the procedure has a hidden implicit parameter (an *environment*). Proc vars that have the calling convention `closure` take up two machine words: One for the proc pointer and another one for the pointer to implicitly passed environment.

stdcall
This the stdcall convention as specified by Microsoft. The generated C procedure is declared with the `__stdcall` keyword.

cdecl
The cdecl convention means that a procedure shall use the same convention as the C compiler. Under windows the generated C procedure is declared with the `__cdecl` keyword.

safecall
This is the safecall convention as specified by Microsoft. The generated C procedure is declared with the `__safecall` keyword. The word *safe* refers to the fact that all hardware registers shall be pushed to the hardware stack.

inline
The inline convention means the the caller should not call the procedure, but inline its code directly. Note that Nim does not inline, but leaves this to the C compiler; it generates `__inline` procedures. This is only a hint for the compiler: it may completely ignore it and it may inline procedures that are not marked as `inline`.

fastcall
Fastcall means different things to different C compilers. One gets whatever the C `__fastcall` means.

syscall
The syscall convention is the same as `__syscall` in C. It is used for interrupts.

noconv
The generated C code will not have any explicit calling convention and thus use the C compiler's default calling convention. This is needed because Nim's default calling convention for procedures is **fastcall** to improve speed.

Most calling conventions exist only for the Windows 32-bit platform.

The default calling convention is **nimcall**, unless it is an inner proc (a proc inside of a proc). For an inner proc an analysis is performed whether it accesses its environment. If it does so, it has the calling convention **closure**, otherwise it has the calling convention **nimcall**.

Distinct type

A **distinct** type is new type derived from a base type that is incompatible with its base type. In particular, it is an essential property of a distinct type that it **does not** imply a subtype relation between it and its base type. Explicit type conversions from a distinct type to its base type and vice versa are allowed.

Modelling currencies

A distinct type can be used to model different physical units with a numerical base type, for example. The following example models currencies.

Different currencies should not be mixed in monetary calculations. Distinct types are a perfect tool to model different currencies:

```
type
  Dollar = distinct int
  Euro = distinct int

var
  d: Dollar
  e: Euro

echo d + 12
# Error: cannot add a number with no unit and a ``Dollar``
```

Unfortunately, **d + 12.Dollar** is not allowed either, because **+** is defined for **int** (among others), not for **Dollar**. So a **+** for dollars needs to be defined:

```
proc `+` (x, y: Dollar): Dollar =
  result = Dollar(int(x) + int(y))
```

It does not make sense to multiply a dollar with a dollar, but with a number without unit; and the same holds for division:

```
proc `*` (x: Dollar, y: int): Dollar =
  result = Dollar(int(x) * y)

proc `*` (x: int, y: Dollar): Dollar =
  result = Dollar(x * int(y))

proc `div` ...
```

This quickly gets tedious. The implementations are trivial and the compiler should not generate all this code only to optimize it away later - after all **+** for dollars should produce the same binary code as **+** for ints. The pragma borrow has been designed to solve this problem; in principle it generates the above trivial implementations:

```
proc `*` (x: Dollar, y: int): Dollar {.borrow.}
proc `*` (x: int, y: Dollar): Dollar {.borrow.}
proc `div` (x: Dollar, y: int): Dollar {.borrow.}
```

The **borrow** pragma makes the compiler use the same implementation as the proc that deals with the distinct type's base type, so no code is generated.

But it seems all this boilerplate code needs to be repeated for the **Euro** currency. This can be solved with templates.

```
template additive(typ: typedesc) =
  proc `+` *(x, y: typ): typ {.borrow.}
  proc `-` *(x, y: typ): typ {.borrow.}

  # unary operators:
  proc `+` *(x: typ): typ {.borrow.}
  proc `-` *(x: typ): typ {.borrow.}

template multiplicative(typ, base: typedesc) =
  proc `*` *(x: typ, y: base): typ {.borrow.}
  proc `*` *(x: base, y: typ): typ {.borrow.}
  proc `div` *(x: typ, y: base): typ {.borrow.}
  proc `mod` *(x: typ, y: base): typ {.borrow.}

template comparable(typ: typedesc) =
  proc `<` * (x, y: typ): bool {.borrow.}
  proc `<=` * (x, y: typ): bool {.borrow.}
  proc `==` * (x, y: typ): bool {.borrow.}

template defineCurrency(typ, base: untyped) =
  type
    typ* = distinct base
  additive(typ)
  multiplicative(typ, base)
  comparable(typ)

defineCurrency(Dollar, int)
defineCurrency(Euro, int)
```

The borrow pragma can also be used to annotate the distinct type to allow certain builtin operations to be lifted:

```
type
  Foo = object
    a, b: int
    s: string

  Bar {.borrow: `.`.} = distinct Foo

var bb: ref Bar
new bb
# field access now valid
bb.a = 90
bb.s = "abc"
```

Currently only the dot accessor can be borrowed in this way.

Avoiding SQL injection attacks

An SQL statement that is passed from Nim to an SQL database might be modelled as a string. However, using string templates and filling in the values is vulnerable to the famous SQL injection attack:

```
import strutils

proc query(db: DbHandle, statement: string) = ...

var
  username: string

db.query("SELECT FROM users WHERE name = '$1'" % username)
# Horrible security hole, but the compiler does not mind!
```

This can be avoided by distinguishing strings that contain SQL from strings that don't. Distinct types provide a means to introduce a new string type **SQL** that is incompatible with **string**:

```
type
  SQL = distinct string

proc query(db: DbHandle, statement: SQL) = ...

var
  username: string

db.query("SELECT FROM users WHERE name = '$1'" % username)
# Error at compile time: `query` expects an SQL string!
```

It is an essential property of abstract types that they **do not** imply a subtype relation between the abstract type and its base type. Explicit type conversions from **string** to **SQL** are allowed:

```
import strutils, sequtils

proc properQuote(s: string): SQL =
  # quotes a string properly for an SQL statement
  return SQL(s)

proc `%` (frmt: SQL, values: openarray[string]): SQL =
  # quote each argument:
  let v = values.mapIt(SQL, properQuote(it))
  # we need a temporary type for the type conversion :-(
  type StrSeq = seq[string]
  # call strutils.`%`:
  result = SQL(string(frmt) % StrSeq(v))

db.query("SELECT FROM users WHERE name = '$1'".SQL % [username])
```

Now we have compile-time checking against SQL injection attacks. Since **"".SQL** is transformed to **SQL("")** no new syntax is needed for nice looking **SQL** string literals. The hypothetical **SQL** type actually exists in the library as the TSqlQuery type (db_sqlite.html#TSqlQuery) of modules like db_sqlite (db_sqlite.html).

Void type

The **void** type denotes the absence of any type. Parameters of type **void** are treated as non-existent, **void** as a return type means that the procedure does not return a value:

```
proc nothing(x, y: void): void =
  echo "ha"

nothing() # writes "ha" to stdout
```

The **void** type is particularly useful for generic code:

```
proc callProc[T](p: proc (x: T), x: T) =
  when T is void:
    p()
  else:
    p(x)

proc intProc(x: int) = discard
proc emptyProc() = discard

callProc[int](intProc, 12)
callProc[void](emptyProc)
```

However, a **void** type cannot be inferred in generic code:

```
callProc(emptyProc)
# Error: type mismatch: got (proc ())
# but expected one of:
# callProc(p: proc (T), x: T)
```

The **void** type is only valid for parameters and return types; other symbols cannot have the type **void**.

Auto type

The **auto** type can only be used for return types and parameters. For return types it causes the compiler to infer the type from the routine body:

```
proc returnsInt(): auto = 1984
```

For parameters it currently creates implicitly generic routines:

```
proc foo(a, b: auto) = discard
```

Is the same as:

```
proc foo[T1, T2](a: T1, b: T2) = discard
```

However later versions of the language might change this to mean "infer the parameters' types from the body". Then the above **foo** would be rejected as the parameters' types can not be inferred from an empty **discard** statement.

Type relations

The following section defines several relations on types that are needed to describe the type checking done by the compiler.

Type equality

Nim uses structural type equivalence for most types. Only for objects, enumerations and distinct types name equivalence is used. The following algorithm, *in pseudo-code*, determines type equality:

```
proc typeEqualsAux(a, b: PType,
                   s: var HashSet[(PType, PType)]): bool =
  if (a,b) in s: return true
  incl(s, (a,b))
  if a.kind == b.kind:
    case a.kind
    of int, intXX, float, floatXX, char, string, cstring, pointer,
        bool, nil, void:
      # leaf type: kinds identical; nothing more to check
      result = true
    of ref, ptr, var, set, seq, openarray:
      result = typeEqualsAux(a.baseType, b.baseType, s)
    of range:
      result = typeEqualsAux(a.baseType, b.baseType, s) and
        (a.rangeA == b.rangeA) and (a.rangeB == b.rangeB)
    of array:
      result = typeEqualsAux(a.baseType, b.baseType, s) and
               typeEqualsAux(a.indexType, b.indexType, s)
    of tuple:
      if a.tupleLen == b.tupleLen:
        for i in 0..a.tupleLen-1:
          if not typeEqualsAux(a[i], b[i], s): return false
        result = true
    of object, enum, distinct:
      result = a == b
    of proc:
      result = typeEqualsAux(a.parameterTuple, b.parameterTuple, s) and
               typeEqualsAux(a.resultType, b.resultType, s) and
               a.callingConvention == b.callingConvention

proc typeEquals(a, b: PType): bool =
  var s: HashSet[(PType, PType)] = {}
  result = typeEqualsAux(a, b, s)
```

Since types are graphs which can have cycles, the above algorithm needs an auxiliary set `s` to detect this case.

Type equality modulo type distinction

The following algorithm (in pseudo-code) determines whether two types are equal with no respect to `distinct` types. For brevity the cycle check with an auxiliary set `s` is omitted:

```
proc typeEqualsOrDistinct(a, b: PType): bool =
  if a.kind == b.kind:
    case a.kind
    of int, intXX, float, floatXX, char, string, cstring, pointer,
        bool, nil, void:
      # leaf type: kinds identical; nothing more to check
      result = true
    of ref, ptr, var, set, seq, openarray:
      result = typeEqualsOrDistinct(a.baseType, b.baseType)
    of range:
      result = typeEqualsOrDistinct(a.baseType, b.baseType) and
        (a.rangeA == b.rangeA) and (a.rangeB == b.rangeB)
    of array:
      result = typeEqualsOrDistinct(a.baseType, b.baseType) and
               typeEqualsOrDistinct(a.indexType, b.indexType)
    of tuple:
      if a.tupleLen == b.tupleLen:
        for i in 0..a.tupleLen-1:
          if not typeEqualsOrDistinct(a[i], b[i]): return false
        result = true
    of distinct:
      result = typeEqualsOrDistinct(a.baseType, b.baseType)
    of object, enum:
      result = a == b
    of proc:
      result = typeEqualsOrDistinct(a.parameterTuple, b.parameterTuple) and
               typeEqualsOrDistinct(a.resultType, b.resultType) and
               a.callingConvention == b.callingConvention
  elif a.kind == distinct:
    result = typeEqualsOrDistinct(a.baseType, b)
  elif b.kind == distinct:
    result = typeEqualsOrDistinct(a, b.baseType)
```

Subtype relation

If object **a** inherits from **b**, **a** is a subtype of **b**. This subtype relation is extended to the types **var**, **ref**, **ptr**:

```
proc isSubtype(a, b: PType): bool =
  if a.kind == b.kind:
    case a.kind
    of object:
      var aa = a.baseType
      while aa != nil and aa != b: aa = aa.baseType
      result = aa == b
    of var, ref, ptr:
      result = isSubtype(a.baseType, b.baseType)
```

Covariance

Covariance in Nim can be introduced only though pointer-like types such as **ptr** and **ref**. Sequence, Array and OpenArray types, instantiated with pointer-like types will be considered covariant if and only if they are also immutable. The introduction of a **var** modifier or additional **ptr** or **ref** indirections would result in invariant treatment of these types.

proc types are currently always invariant, but future versions of Nim may relax this rule.

User-defined generic types may also be covariant with respect to some of their parameters. By default, all generic params are considered invariant, but you may choose the apply the prefix modifier **in** to a parameter to make it contravariant or **out** to make it covariant:

```
type
  AnnotatedPtr[out T] =
    metadata: MyTypeInfo
    p: ref T

  RingBuffer[out T] =
    startPos: int
    data: seq[T]

  Action {.importcpp: "std::function<void ('0)>".} [in T] = object
```

When the designated generic parameter is used to instantiate a pointer-like type as in the case of *AnnotatedPtr* above, the resulting generic type will also have pointer-like covariance:

```
type
  GuiWidget = object of RootObj
  Button = object of GuiWidget
  ComboBox = object of GuiWidget

var
  widgetPtr: AnnotatedPtr[GuiWidget]
  buttonPtr: AnnotatedPtr[Button]

...

proc drawWidget[T](x: AnnotatedPtr[GuiWidget]) = ...

# you can call procs expecting base types by supplying a derived type
drawWidget(buttonPtr)

# and you can convert more-specific pointer types to more general ones
widgetPtr = buttonPtr
```

Just like with regular pointers, covariance will be enabled only for immutable values:

```
proc makeComboBox[T](x: var AnnotatedPtr[GuiWidget]) =
  x.p = new(ComboBox)

makeComboBox(buttonPtr) # Error, AnnotatedPtr[Button] cannot be modified
                        # to point to a ComboBox
```

On the other hand, in the *RingBuffer* example above, the designated generic param is used to instantiate the non-pointer **seq** type, which means that the resulting generic type will have covariance that mimics an array or sequence (i.e. it will be covariant only when instantiated with **ptr** and **ref** types):

```
type
  Base = object of RootObj
  Derived = object of Base

proc consumeBaseValues(b: RingBuffer[Base]) = ...

var derivedValues: RingBuffer[Derived]

consumeBaseValues(derivedValues) # Error, Base and Derived values may differ
                                 # in size

proc consumeBasePointers(b: RingBuffer[ptr Base]) = ...

var derivedPointers: RingBuffer[ptr Derived]

consumeBaseValues(derivedPointers) # This is legal
```

Please note that Nim will treat the user-defined pointer-like types as proper alternatives to the built-in pointer types. That is, types such as *seq[AnnotatedPtr[T]]* or *RingBuffer[AnnotatedPtr[T]]* will also be considered covariant and you can create new pointer-like types by instantiating other user-defined pointer-like types.

The contravariant parameters introduced with the **in** modifier are currently useful only when interfacing with imported types having such semantics.

Convertible relation

A type **a** is **implicitly** convertible to type **b** iff the following algorithm returns true:

```
# XXX range types?
proc isImplicitlyConvertible(a, b: PType): bool =
  if isSubtype(a, b) or isCovariant(a, b):
    return true
  case a.kind
  of int:     result = b in {int8, int16, int32, int64, uint, uint8, uint16,
                             uint32, uint64, float, float32, float64}
  of int8:    result = b in {int16, int32, int64, int}
  of int16:   result = b in {int32, int64, int}
  of int32:   result = b in {int64, int}
  of uint:    result = b in {uint32, uint64}
  of uint8:   result = b in {uint16, uint32, uint64}
  of uint16:  result = b in {uint32, uint64}
  of uint32:  result = b in {uint64}
  of float:   result = b in {float32, float64}
  of float32: result = b in {float64, float}
  of float64: result = b in {float32, float}
  of seq:
    result = b == openArray and typeEquals(a.baseType, b.baseType)
  of array:
    result = b == openArray and typeEquals(a.baseType, b.baseType)
    if a.baseType == char and a.indexType.rangeA == 0:
      result = b = cstring
  of cstring, ptr:
    result = b == pointer
  of string:
    result = b == cstring
```

A type **a** is **explicitly** convertible to type **b** iff the following algorithm returns true:

```
proc isIntegralType(t: PType): bool =
  result = isOrdinal(t) or t.kind in {float, float32, float64}

proc isExplicitlyConvertible(a, b: PType): bool =
  result = false
  if isImplicitlyConvertible(a, b): return true
  if typeEqualsOrDistinct(a, b): return true
  if isIntegralType(a) and isIntegralType(b): return true
  if isSubtype(a, b) or isSubtype(b, a): return true
```

The convertible relation can be relaxed by a user-defined type converter.

```
converter toInt(x: char): int = result = ord(x)

var
  x: int
  chr: char = 'a'

# implicit conversion magic happens here
x = chr
echo x # => 97
# you can use the explicit form too
x = chr.toInt
echo x # => 97
```

The type conversion **`T(a)`** is an L-value if **`a`** is an L-value and **`typeEqualsOrDistinct(T, type(a))`** holds.

Assignment compatibility

An expression **`b`** can be assigned to an expression **`a`** iff **`a`** is an *l-value* and **`isImplicitlyConvertible(b.typ, a.typ)`** holds.

Overloading resolution

In a call **p(args)** the routine **p** that matches best is selected. If multiple routines match equally well, the ambiguity is reported at compiletime.

Every arg in args needs to match. There are multiple different categories how an argument can match. Let **f** be the formal parameter's type and **a** the type of the argument.

1. Exact match: **a** and **f** are of the same type.
2. Literal match: **a** is an integer literal of value **v** and **f** is a signed or unsigned integer type and **v** is in **f**'s range. Or: **a** is a floating point literal of value **v** and **f** is a floating point type and **v** is in **f**'s range.
3. Generic match: **f** is a generic type and **a** matches, for instance **a** is **int** and **f** is a generic (constrained) parameter type (like in **[T]** or **[T: int|char]**.
4. Subrange or subtype match: **a** is a **range[T]** and **T** matches **f** exactly. Or: **a** is a subtype of **f**.
5. Integral conversion match: **a** is convertible to **f** and **f** and **a** is some integer or floating point type.
6. Conversion match: **a** is convertible to **f**, possibly via a user defined **converter**.

These matching categories have a priority: An exact match is better than a literal match and that is better than a generic match etc. In the following **count(p, m)** counts the number of matches of the matching category **m** for the routine **p**.

A routine **p** matches better than a routine **q** if the following algorithm returns true:

```
for each matching category m in ["exact match", "literal match",
                                "generic match", "subtype match",
                                "integral match", "conversion match"]:
  if count(p, m) > count(q, m): return true
  elif count(p, m) == count(q, m):
    discard "continue with next category m"
  else:
    return false
return "ambiguous"
```

Some examples:

```
proc takesInt(x: int) = echo "int"
proc takesInt[T](x: T) = echo "T"
proc takesInt(x: int16) = echo "int16"

takesInt(4) # "int"
var x: int32
takesInt(x) # "T"
var y: int16
takesInt(y) # "int16"
var z: range[0..4] = 0
takesInt(z) # "T"
```

If this algorithm returns "ambiguous" further disambiguation is performed: If the argument **a** matches both the parameter type **f** of **p** and **g** of **q** via a subtyping relation, the inheritance depth is taken into account:

```
type
  A = object of RootObj
  B = object of A
  C = object of B

proc p(obj: A) =
  echo "A"

proc p(obj: B) =
  echo "B"

var c = C()
# not ambiguous, calls 'B', not 'A' since B is a subtype of A
# but not vice versa:
p(c)

proc pp(obj: A, obj2: B) = echo "A B"
proc pp(obj: B, obj2: A) = echo "B A"

# but this is ambiguous:
pp(c, c)
```

Likewise for generic matches the most specialized generic type (that still matches) is preferred:

```
proc gen[T](x: ref ref T) = echo "ref ref T"
proc gen[T](x: ref T) = echo "ref T"
proc gen[T](x: T) = echo "T"

var ri: ref int
gen(ri) # "ref T"
```

Overloading based on 'var T'

If the formal parameter **f** is of type **var T** in addition to the ordinary type checking, the argument is checked to be an l-value. **var T** matches better than just **T** then.

```
proc sayHi(x: int): string =
  # matches a non-var int
  result = $x
proc sayHi(x: var int): string =
  # matches a var int
  result = $(x + 10)

proc sayHello(x: int) =
  var m = x # a mutable version of x
  echo sayHi(x) # matches the non-var version of sayHi
  echo sayHi(m) # matches the var version of sayHi

sayHello(3) # 3
            # 13
```

Automatic dereferencing

If the experimental mode is active and no other match is found, the first argument **a** is dereferenced automatically if it's a pointer type and overloading resolution is tried with **a[]**

Automatic self insertions

Starting with version 0.14 of the language, Nim supports **`field`** as a shortcut for **`self.field`** comparable to the this keyword in Java or C++. This feature has to be explicitly enabled via a **`{.this: self.}`** statement pragma. This pragma is active for the rest of the module:

```
type
  Parent = object of RootObj
    parentField: int
  Child = object of Parent
    childField: int

{.this: self.}
proc sumFields(self: Child): int =
  result = parentField + childField
  # is rewritten to:
  # result = self.parentField + self.childField
```

Instead of **`self`** any other identifier can be used too, but **`{.this: self.}`** will become the default directive for the whole language eventually.

In addition to fields, routine applications are also rewritten, but only if no other interpretation of the call is possible:

```
proc test(self: Child) =
  echo childField, " ", sumFields()
  # is rewritten to:
  echo self.childField, " ", sumFields(self)
  # but NOT rewritten to:
  echo self, self.childField, " ", sumFields(self)
```

Lazy type resolution for untyped

Note: An unresolved expression is an expression for which no symbol lookups and no type checking have been performed.

Since templates and macros that are not declared as **`immediate`** participate in overloading resolution it's essential to have a way to pass unresolved expressions to a template or macro. This is what the meta-type **`untyped`** accomplishes:

```
template rem(x: untyped) = discard

rem unresolvedExpression(undeclaredIdentifier)
```

A parameter of type **`untyped`** always matches any argument (as long as there is any argument passed to it).

But one has to watch out because other overloads might trigger the argument's resolution:

```
template rem(x: untyped) = discard
proc rem[T](x: T) = discard

# undeclared identifier: 'unresolvedExpression'
rem unresolvedExpression(undeclaredIdentifier)
```

`untyped` and **`varargs[untyped]`** are the only metatype that are lazy in this sense, the other metatypes **`typed`** and **`typedesc`** are not lazy.

Varargs matching

See Varargs.

Statements and expressions

Nim uses the common statement/expression paradigm: Statements do not produce a value in contrast to expressions. However, some expressions are statements.

Statements are separated into simple statements and complex statements. Simple statements are statements that cannot contain other statements like assignments, calls or the **`return`** statement; complex statements can contain other statements. To avoid the dangling else problem, complex statements always have to be indented. The details can be found in the grammar.

Statement list expression

Statements can also occur in an expression context that looks like **`(stmt1; stmt2; ...; ex)`**. This is called an statement list expression or **`(;)`**. The type of **`(stmt1; stmt2; ...; ex)`** is the type of **`ex`**. All the other statements must be of type **`void`**. (One can use **`discard`** to produce a **`void`** type.) **`(;)`** does not introduce a new scope.

Discard statement

Example:

```
proc p(x, y: int): int =
  result = x + y

discard p(3, 4) # discard the return value of `p`
```

The **`discard`** statement evaluates its expression for side-effects and throws the expression's resulting value away.

Ignoring the return value of a procedure without using a discard statement is a static error.

The return value can be ignored implicitly if the called proc/iterator has been declared with the discardable pragma:

```
proc p(x, y: int): int {.discardable.} =
  result = x + y

p(3, 4) # now valid
```

An empty **`discard`** statement is often used as a null statement:

```
proc classify(s: string) =
  case s[0]
  of SymChars, '_': echo "an identifier"
  of '0'..'9': echo "a number"
  else: discard
```

Void context

In a list of statements every expression except the last one needs to have the type **`void`**. In addition to this rule an assignment to the builtin **`result`** symbol also triggers a mandatory **`void`** context for the subsequent expressions:

```
proc invalid*(): string =
  result = "foo"
  "invalid"  # Error: value of type 'string' has to be discarded
```

```
proc valid*(): string =
  let x = 317
  "valid"
```

Var statement

Var statements declare new local and global variables and initialize them. A comma separated list of variables can be used to specify variables of the same type:

```
var
  a: int = 0
  x, y, z: int
```

If an initializer is given the type can be omitted: the variable is then of the same type as the initializing expression. Variables are always initialized with a default value if there is no initializing expression. The default value depends on the type and is always a zero in binary.

Type	default value
any integer type	0
any float	0.0
char	'\0'
bool	false
ref or pointer type	nil
procedural type	nil
sequence	nil (*not* `@[]`)
string	nil (*not* "")
tuple[x: A, y: B, ...]	(default(A), default(B), ...) (analogous for objects)
array[0..., T]	[default(T), ...]
range[T]	default(T); this may be out of the valid range
T = enum	cast[T](0); this may be an invalid value

The implicit initialization can be avoided for optimization reasons with the noinit pragma:

```
var
  a {.noInit.}: array [0..1023, char]
```

If a proc is annotated with the **`noinit`** pragma this refers to its implicit **`result`** variable:

```
proc returnUndefinedValue: int {.noinit.} = discard
```

The implicit initialization can be also prevented by the requiresInit type pragma. The compiler requires an explicit initialization for the object and all of its fields. However it does a control flow analysis to prove the variable has been initialized and does not rely on syntactic properties:

```
type
  MyObject = object {.requiresInit.}

proc p() =
  # the following is valid:
  var x: MyObject
  if someCondition():
    x = a()
  else:
    x = a()
  use x
```

let statement

A **`let`** statement declares new local and global single assignment variables and binds a value to them. The syntax is the same as that of the **`var`** statement, except that the keyword **`var`** is replaced by the keyword **`let`**. Let variables are not l-values and can thus not be passed to **`var`** parameters nor can their address be taken. They cannot be assigned new values.

For let variables the same pragmas are available as for ordinary variables.

Tuple unpacking

In a **`var`** or **`let`** statement tuple unpacking can be performed. The special identifier **`_`** can be used to ignore some parts of the tuple:

```
proc returnsTuple(): (int, int, int) = (4, 2, 3)

let (x, _, z) = returnsTuple()
```

Const section

Constants are symbols which are bound to a value. The constant's value cannot change. The compiler must be able to evaluate the expression in a constant declaration at compile time.

Nim contains a sophisticated compile-time evaluator, so procedures which have no side-effect can be used in constant expressions too:

```
import strutils
const
  constEval = contains("abc", 'b') # computed at compile time!
```

The rules for compile-time computability are:

1. Literals are compile-time computable.
2. Type conversions are compile-time computable.
3. Procedure calls of the form **`p(X)`** are compile-time computable if **`p`** is a proc without side-effects (see the noSideEffect pragma for details) and if **`X`** is a (possibly empty) list of compile-time computable arguments.

Constants cannot be of type **`ptr`**, **`ref`**, **`var`** or **`object`**, nor can they contain such a type.

Static statement/expression

A static statement/expression can be used to enforce compile time evaluation explicitly. Enforced compile time evaluation can even evaluate code that has side effects:

```
static:
  echo "echo at compile time"
```

It's a static error if the compiler cannot perform the evaluation at compile time.

The current implementation poses some restrictions for compile time evaluation: Code which contains **cast** or makes use of the foreign function interface cannot be evaluated at compile time. Later versions of Nim will support the FFI at compile time.

If statement

Example:

```
var name = readLine(stdin)

if name == "Andreas":
  echo "What a nice name!"
elif name == "":
  echo "Don't you have a name?"
else:
  echo "Boring name..."
```

The **if** statement is a simple way to make a branch in the control flow: The expression after the keyword **if** is evaluated, if it is true the corresponding statements after the **:** are executed. Otherwise the expression after the **elif** is evaluated (if there is an **elif** branch), if it is true the corresponding statements after the **:** are executed. This goes on until the last **elif**. If all conditions fail, the **else** part is executed. If there is no **else** part, execution continues with the next statement.

In **if** statements new scopes begin immediately after the **if/elif/else** keywords and ends after the corresponding *then* block. For visualization purposes the scopes have been enclosed in **{| |}** in the following example:

```
if {| (let m = input =~ re"(\w+)=\w+"; m.isMatch):
  echo "key ", m[0], " value ", m[1]  |}
elif {| (let m = input =~ re""; m.isMatch):
  echo "new m in this scope"  |}
else: {|
  echo "m not declared here"  |}
```

Case statement

Example:

```
case readline(stdin)
of "delete-everything", "restart-computer":
  echo "permission denied"
of "go-for-a-walk":     echo "please yourself"
else:                   echo "unknown command"

# indentation of the branches is also allowed; and so is an optional colon
# after the selecting expression:
case readline(stdin):
  of "delete-everything", "restart-computer":
    echo "permission denied"
  of "go-for-a-walk":     echo "please yourself"
  else:                   echo "unknown command"
```

The **case** statement is similar to the if statement, but it represents a multi-branch selection. The expression after the keyword **case** is evaluated and if its value is in a *slicelist* the corresponding statements (after the **of** keyword) are executed. If the value is not in any given *slicelist* the **else** part is executed. If there is no **else** part and not all possible values that **expr** can hold occur in a **slicelist**, a static error occurs. This holds only for expressions of ordinal types. "All possible values" of **expr** are determined by **expr**'s type. To suppress the static error an **else** part with an empty **discard** statement should be used.

For non ordinal types it is not possible to list every possible value and so these always require an **else** part.

As a special semantic extension, an expression in an **of** branch of a case statement may evaluate to a set or array constructor; the set or array is then expanded into a list of its elements:

```
const
  SymChars: set[char] = {'a'..'z', 'A'..'Z', '\x80'..'\xFF'}

proc classify(s: string) =
  case s[0]
  of SymChars, '_': echo "an identifier"
  of '0'..'9': echo "a number"
  else: echo "other"

# is equivalent to:
proc classify(s: string) =
  case s[0]
  of 'a'..'z', 'A'..'Z', '\x80'..'\xFF', '_': echo "an identifier"
  of '0'..'9': echo "a number"
  else: echo "other"
```

When statement

Example:

```
when sizeof(int) == 2:
  echo "running on a 16 bit system!"
elif sizeof(int) == 4:
  echo "running on a 32 bit system!"
elif sizeof(int) == 8:
  echo "running on a 64 bit system!"
else:
  echo "cannot happen!"
```

The **when** statement is almost identical to the **if** statement with some exceptions:

- Each condition (**expr**) has to be a constant expression (of type **bool**).
- The statements do not open a new scope.
- The statements that belong to the expression that evaluated to true are translated by the compiler, the other statements are not checked for semantics! However, each condition is checked for semantics.

The **when** statement enables conditional compilation techniques. As a special syntactic extension, the **when** construct is also available within **object** definitions.

When nimvm statement

nimvm is a special symbol, that may be used as expression of **when nimvm** statement to differentiate execution path between runtime and compile time.

Example:

```
proc someProcThatMayRunInCompileTime(): bool =
  when nimvm:
    # This code runs in compile time
    result = true
  else:
    # This code runs in runtime
    result = false
const ctValue = someProcThatMayRunInCompileTime()
let rtValue = someProcThatMayRunInCompileTime()
assert(ctValue == true)
assert(rtValue == false)
```

when nimvm statement must meet the following requirements:

- Its expression must always be **nimvm**. More complex expressions are not allowed.
- It must not contain **elif** branches.
- It must contain **else** branch.
- Code in branches must not affect semantics of the code that follows the **when nimvm** statement. E.g. it must not define symbols that are used in the following code.

Return statement

Example:

```
return 40+2
```

The **return** statement ends the execution of the current procedure. It is only allowed in procedures. If there is an **expr**, this is syntactic sugar for:

```
result = expr
return result
```

return without an expression is a short notation for **return result** if the proc has a return type. The result variable is always the return value of the procedure. It is automatically declared by the compiler. As all variables, **result** is initialized to (binary) zero:

```
proc returnZero(): int =
  # implicitly returns 0
```

Yield statement

Example:

```
yield (1, 2, 3)
```

The **yield** statement is used instead of the **return** statement in iterators. It is only valid in iterators. Execution is returned to the body of the for loop that called the iterator. Yield does not end the iteration process, but execution is passed back to the iterator if the next iteration starts. See the section about iterators (Iterators and the for statement) for further information.

Block statement

Example:

```
var found = false
block myblock:
  for i in 0..3:
    for j in 0..3:
      if a[j][i] == 7:
        found = true
        break myblock # leave the block, in this case both for-loops
echo found
```

The block statement is a means to group statements to a (named) **block**. Inside the block, the **break** statement is allowed to leave the block immediately. A **break** statement can contain a name of a surrounding block to specify which block is to leave.

Break statement

Example:

```
break
```

The **break** statement is used to leave a block immediately. If **symbol** is given, it is the name of the enclosing block that is to leave. If it is absent, the innermost block is left.

While statement

Example:

```
echo "Please tell me your password:"
var pw = readLine(stdin)
while pw != "12345":
  echo "Wrong password! Next try:"
  pw = readLine(stdin)
```

The **while** statement is executed until the **expr** evaluates to false. Endless loops are no error. **while** statements open an *implicit block*, so that they can be left with a **break** statement.

Continue statement

A **continue** statement leads to the immediate next iteration of the surrounding loop construct. It is only allowed within a loop. A continue statement is syntactic sugar for a nested block:

```
while expr1:
  stmt1
  continue
  stmt2
```

Is equivalent to:

```
while expr1:
  block myBlockName:
    stmt1
    break myBlockName
    stmt2
```

Assembler statement

The direct embedding of assembler code into Nim code is supported by the unsafe **asm** statement. Identifiers in the assembler code that refer to Nim identifiers shall be enclosed in a special character which can be specified in the statement's pragmas. The default special character is '`':

```
{.push stackTrace:off.}
proc addInt(a, b: int): int =
  # a in eax, and b in edx
  asm """
      mov eax, `a`
      add eax, `b`
      jno theEnd
      call `raiseOverflow`
    theEnd:
  """
{.pop.}
```

If the GNU assembler is used, quotes and newlines are inserted automatically:

```
proc addInt(a, b: int): int =
  asm """
    addl %%ecx, %%eax
    jno 1
    call `raiseOverflow`
    1:
    :"=a"(`result`)
    :"a"(`a`), "c"(`b`)
  """
```

Instead of:

```
proc addInt(a, b: int): int =
  asm """
    "addl %%ecx, %%eax\n"
    "jno 1\n"
    "call `raiseOverflow`\n"
    "1: \n"
    :"=a"(`result`)
    :"a"(`a`), "c"(`b`)
  """
```

Using statement

Warning: The **using** statement is experimental and has to be explicitly enabled with the experimental pragma or command line option!

The using statement provides syntactic convenience in modules where the same parameter names and types are used over and over. Instead of:

```
proc foo(c: Context; n: Node) = ...
proc bar(c: Context; n: Node, counter: int) = ...
proc baz(c: Context; n: Node) = ...
```

One can tell the compiler about the convention that a parameter of name **c** should default to type **Context**, **n** should default to **Node** etc.:

```
{.experimental.}
using
  c: Context
  n: Node
  counter: int

proc foo(c, n) = ...
proc bar(c, n, counter) = ...
proc baz(c, n) = ...
```

The **using** section uses the same indentation based grouping syntax as a **var** or **let** section.

Note that **using** is not applied for **template** since untyped template parameters default to the type **system.untyped**.

If expression

An *if expression* is almost like an if statement, but it is an expression. Example:

```
var y = if x > 8: 9 else: 10
```

An if expression always results in a value, so the **else** part is required. **Elif** parts are also allowed.

When expression

Just like an *if expression*, but corresponding to the when statement.

Case expression

The *case expression* is again very similar to the case statement:

```
var favoriteFood = case animal
  of "dog": "bones"
  of "cat": "mice"
  elif animal.endsWith"whale": "plankton"
  else:
    echo "I'm not sure what to serve, but everybody loves ice cream"
    "ice cream"
```

As seen in the above example, the case expression can also introduce side effects. When multiple statements are given for a branch, Nim will use the last expression as the result value, much like in an *expr* template.

Table constructor

A table constructor is syntactic sugar for an array constructor:

```
{"key1": "value1", "key2", "key3": "value2"}

# is the same as:
[("key1", "value1"), ("key2", "value2"), ("key3", "value2")]
```

The empty table can be written **`{:}`** (in contrast to the empty set which is **`{}`**) which is thus another way to write as the empty array constructor **`[]`**. This slightly unusual way of supporting tables has lots of advantages:

- The order of the (key,value)-pairs is preserved, thus it is easy to support ordered dicts with for example **`{key: val}.newOrderedTable`**.
- A table literal can be put into a **`const`** section and the compiler can easily put it into the executable's data section just like it can for arrays and the generated data section requires a minimal amount of memory.
- Every table implementation is treated equal syntactically.
- Apart from the minimal syntactic sugar the language core does not need to know about tables.

Type conversions

Syntactically a *type conversion* is like a procedure call, but a type name replaces the procedure name. A type conversion is always safe in the sense that a failure to convert a type to another results in an exception (if it cannot be determined statically).

Ordinary procs are often preferred over type conversions in Nim: For instance, **`$`** is the **`toString`** operator by convention and **`toFloat`** and **`toInt`** can be used to convert from floating point to integer or vice versa.

Type casts

Example:

```
cast[int](x)
```

Type casts are a crude mechanism to interpret the bit pattern of an expression as if it would be of another type. Type casts are only needed for low-level programming and are inherently unsafe.

The addr operator

The **`addr`** operator returns the address of an l-value. If the type of the location is **`T`**, the *addr* operator result is of the type **`ptr T`**. An address is always an untraced reference. Taking the address of an object that resides on the stack is **unsafe**, as the pointer may live longer than the object on the stack and can thus reference a non-existing object. One can get the address of variables, but one can't use it on variables declared through **`let`** statements:

```
let t1 = "Hello"
var
  t2 = t1
  t3 : pointer = addr(t2)
echo repr(addr(t2))
# --> ref 0x7fff6b71b670 --> 0x10bb81050"Hello"
echo cast[ptr string](t3)[]
# --> Hello
# The following line doesn't compile:
echo repr(addr(t1))
# Error: expression has no address
```

Procedures

What most programming languages call methods or functions are called procedures in Nim. A procedure declaration consists of an identifier, zero or more formal parameters, a return value type and a block of code. Formal parameters are declared as a list of identifiers separated by either comma or semicolon. A parameter is given a type by `: typename`. The type applies to all parameters immediately before it, until either the beginning of the parameter list, a semicolon separator or an already typed parameter, is reached. The semicolon can be used to make separation of types and subsequent identifiers more distinct.

```
# Using only commas
proc foo(a, b: int, c, d: bool): int

# Using semicolon for visual distinction
proc foo(a, b: int; c, d: bool): int

# Will fail: a is untyped since ';' stops type propagation.
proc foo(a; b: int; c, d: bool): int
```

A parameter may be declared with a default value which is used if the caller does not provide a value for the argument.

```
# b is optional with 47 as its default value
proc foo(a: int, b: int = 47): int
```

Parameters can be declared mutable and so allow the proc to modify those arguments, by using the type modifier *var*.

```
# "returning" a value to the caller through the 2nd argument
# Notice that the function uses no actual return value at all (ie void)
proc foo(inp: int, outp: var int) =
  outp = inp + 47
```

If the proc declaration has no body, it is a forward declaration. If the proc returns a value, the procedure body can access an implicitly declared variable named result that represents the return value. Procs can be overloaded. The overloading resolution algorithm determines which proc is the best match for the arguments. Example:

```
proc toLower(c: char): char = # toLower for characters
  if c in {'A'..'Z'}:
    result = chr(ord(c) + (ord('a') - ord('A')))
  else:
    result = c

proc toLower(s: string): string = # toLower for strings
  result = newString(len(s))
  for i in 0..len(s) - 1:
    result[i] = toLower(s[i]) # calls toLower for characters; no recursion!
```

Calling a procedure can be done in many different ways:

```
proc callme(x, y: int, s: string = "", c: char, b: bool = false) = ...

# call with positional arguments      # parameter bindings:
callme(0, 1, "abc", '\t', true)       # (x=0, y=1, s="abc", c='\t', b=true)
# call with named and positional arguments:
callme(y=1, x=0, "abd", '\t')         # (x=0, y=1, s="abd", c='\t', b=false)
# call with named arguments (order is not relevant):
callme(c='\t', y=1, x=0)              # (x=0, y=1, s="", c='\t', b=false)
# call as a command statement: no () needed:
callme 0, 1, "abc", '\t'              # (x=0, y=1, s="abc", c='\t', b=false)
```

A procedure may call itself recursively.

Operators are procedures with a special operator symbol as identifier:

```
proc `$` (x: int): string =
  # converts an integer to a string; this is a prefix operator.
  result = intToStr(x)
```

Operators with one parameter are prefix operators, operators with two parameters are infix operators. (However, the parser distinguishes these from the operator's position within an expression.) There is no way to declare postfix operators: all postfix operators are built-in and handled by the grammar explicitly.

Any operator can be called like an ordinary proc with the '*opr*' notation. (Thus an operator can have more than two parameters):

```
proc `*+` (a, b, c: int): int =
  # Multiply and add
  result = a * b + c

assert `*+`(3, 4, 6) == `*`(a, `+`(b, c))
```

Export marker

If a declared symbol is marked with an asterisk it is exported from the current module:

```
proc exportedEcho*(s: string) = echo s
proc `*`*(a: string; b: int): string =
  result = newStringOfCap(a.len * b)
  for i in 1..b: result.add a

var exportedVar*: int
const exportedConst* = 78
type
  ExportedType* = object
    exportedField*: int
```

Method call syntax

For object oriented programming, the syntax **`obj.method(args)`** can be used instead of **`method(obj, args)`**. The parentheses can be omitted if there are no remaining arguments: **`obj.len`** (instead of **`len(obj)`**).

This method call syntax is not restricted to objects, it can be used to supply any type of first argument for procedures:

```
echo "abc".len # is the same as echo len "abc"
echo "abc".toUpper()
echo {'a', 'b', 'c'}.card
stdout.writeLine("Hallo") # the same as writeLine(stdout, "Hallo")
```

Another way to look at the method call syntax is that it provides the missing postfix notation.

The method call syntax conflicts with explicit generic instantiations: `p[T](x)` cannot be written as `x.p[T]` because `x.p[T]` is always parsed as `(x.p)[T]`.

Future directions: `p[.T.]` might be introduced as an alternative syntax to pass explict types to a generic and then `x.p[.T.]` can be parsed as `x.(p[.T.])`.

See also: Limitations of the method call syntax.

Properties

Nim has no need for *get-properties*: Ordinary get-procedures that are called with the *method call syntax* achieve the same. But setting a value is different; for this a special setter syntax is needed:

```
type
  Socket* = ref object of RootObj
    FHost: int # cannot be accessed from the outside of the module
               # the `F` prefix is a convention to avoid clashes since
               # the accessors are named `host`

proc `host=`*(s: var Socket, value: int) {.inline.} =
  ## setter of hostAddr
  s.FHost = value

proc host*(s: Socket): int {.inline.} =
  ## getter of hostAddr
  s.FHost

var s: Socket
new s
s.host = 34  # same as `host=`(s, 34)
```

Command invocation syntax

Routines can be invoked without the `()` if the call is syntatically a statement. This command invocation syntax also works for expressions, but then only a single argument may follow. This restriction means `echo f 1, f 2` is parsed as `echo(f(1), f(2))` and not as `echo(f(1, f(2)))`. The method call syntax may be used to provide one more argument in this case:

```
proc optarg(x: int, y: int = 0): int = x + y
proc singlearg(x: int): int = 20*x

echo optarg 1, " ", singlearg 2  # prints "1 40"

let fail = optarg 1, optarg 8   # Wrong. Too many arguments for a command call
let x = optarg(1, optarg 8)  # traditional procedure call with 2 arguments
let y = 1.optarg optarg 8    # same thing as above, w/o the parenthesis
assert x == y
```

The command invocation syntax also can't have complex expressions as arguments. For example: (anonymous procs), **if**, **case** or **try**. The (do notation) is limited, but usable for a single proc (see the example in the corresponding section). Function calls with no arguments still needs () to distinguish between a call and the function itself as a first class value.

Closures

Procedures can appear at the top level in a module as well as inside other scopes, in which case they are called nested procs. A nested proc can access local variables from its enclosing scope and if it does so it becomes a closure. Any captured variables are stored in a hidden additional argument to the closure (its environment) and they are accessed by reference by both the closure and its enclosing scope (i.e. any modifications made to them are visible in both places). The closure environment may be allocated on the heap or on the stack if the compiler determines that this would be safe.

Creating closures in loops

Since closures capture local variables by reference it is often not wanted behavior inside loop bodies. See closureScope (system.html#closureScope) for details on how to change this behavior.

Anonymous Procs

Procs can also be treated as expressions, in which case it's allowed to omit the proc's name.

```
var cities = @["Frankfurt", "Tokyo", "New York", "Kyiv"]

cities.sort(proc (x,y: string): int =
    cmp(x.len, y.len))
```

Procs as expressions can appear both as nested procs and inside top level executable code.

Do notation

As a special more convenient notation, proc expressions involved in procedure calls can use the **do** keyword:

```
sort(cities) do (x,y: string) -> int:
  cmp(x.len, y.len)

# Less parenthesis using the method plus command syntax:
cities = cities.map do (x:string) -> string:
  "City of " & x

# In macros, the do notation is often used for quasi-quoting
macroResults.add quote do:
  if not `ex`:
    echo `info`, ": Check failed: ", `expString`
```

do is written after the parentheses enclosing the regular proc params. The proc expression represented by the do block is appended to them. In calls using the command syntax, the do block will bind to the immediately preceeding expression, transforming it in a call.

do with parentheses is an anonymous **proc**; however a **do** without parentheses is just a block of code. The **do** notation can be used to pass multiple blocks to a macro:

```
macro performWithUndo(task, undo: untyped) = ...

performWithUndo do:
  # multiple-line block of code
  # to perform the task
do:
  # code to undo it
```

Nonoverloadable builtins

The following builtin procs cannot be overloaded for reasons of implementation simplicity (they require specialized semantic checking):

```
declared, defined, definedInScope, compiles, sizeOf,
is, shallowCopy, getAst, astToStr, spawn, procCall
```

Thus they act more like keywords than like ordinary identifiers; unlike a keyword however, a redefinition may shadow the definition in the **system** module. From this list the following should not be written in dot notation **x.f** since **x** cannot be type checked before it gets passed to **f**:

```
declared, defined, definedInScope, compiles, getAst, astToStr
```

Var parameters

The type of a parameter may be prefixed with the **var** keyword:

```
proc divmod(a, b: int; res, remainder: var int) =
  res = a div b
  remainder = a mod b

var
  x, y: int

divmod(8, 5, x, y) # modifies x and y
assert x == 1
assert y == 3
```

In the example, **res** and **remainder** are *var parameters*. Var parameters can be modified by the procedure and the changes are visible to the caller. The argument passed to a var parameter has to be an l-value. Var parameters are implemented as hidden pointers. The above example is equivalent to:

```
proc divmod(a, b: int; res, remainder: ptr int) =
  res[] = a div b
  remainder[] = a mod b

var
  x, y: int
divmod(8, 5, addr(x), addr(y))
assert x == 1
assert y == 3
```

In the examples, var parameters or pointers are used to provide two return values. This can be done in a cleaner way by returning a tuple:

```
proc divmod(a, b: int): tuple[res, remainder: int] =
  (a div b, a mod b)

var t = divmod(8, 5)

assert t.res == 1
assert t.remainder == 3
```

One can use tuple unpacking to access the tuple's fields:

```
var (x, y) = divmod(8, 5) # tuple unpacking
assert x == 1
assert y == 3
```

Note: `var` parameters are never necessary for efficient parameter passing. Since non-var parameters cannot be modified the compiler is always free to pass arguments by reference if it considers it can speed up execution.

Var return type

A proc, converter or iterator may return a `var` type which means that the returned value is an l-value and can be modified by the caller:

```
var g = 0

proc WriteAccessToG(): var int =
  result = g

WriteAccessToG() = 6
assert g == 6
```

It is a compile time error if the implicitly introduced pointer could be used to access a location beyond its lifetime:

```
proc WriteAccessToG(): var int =
  var g = 0
  result = g # Error!
```

For iterators, a component of a tuple return type can have a `var` type too:

```
iterator mpairs(a: var seq[string]): tuple[key: int, val: var string] =
  for i in 0..a.high:
    yield (i, a[i])
```

In the standard library every name of a routine that returns a `var` type starts with the prefix `m` per convention.

Overloading of the subscript operator

The `[]` subscript operator for arrays/openarrays/sequences can be overloaded.

Multi-methods

Procedures always use static dispatch. Multi-methods use dynamic dispatch. For dynamic dispatch to work on an object it should be a reference type as well.

```
type
  Expression = ref object of RootObj ## abstract base class for an expression
  Literal = ref object of Expression
    x: int
  PlusExpr = ref object of Expression
    a, b: Expression

method eval(e: Expression): int {.base.} =
  # override this base method
  quit "to override!"

method eval(e: Literal): int = return e.x

method eval(e: PlusExpr): int =
  # watch out: relies on dynamic binding
  result = eval(e.a) + eval(e.b)

proc newLit(x: int): Literal =
  new(result)
  result.x = x

proc newPlus(a, b: Expression): PlusExpr =
  new(result)
  result.a = a
  result.b = b

echo eval(newPlus(newPlus(newLit(1), newLit(2)), newLit(4)))
```

In the example the constructors **`newLit`** and **`newPlus`** are procs because they should use static binding, but **`eval`** is a method because it requires dynamic binding.

As can be seen in the example, base methods have to be annotated with the base pragma. The **`base`** pragma also acts as a reminder for the programmer that a base method **`m`** is used as the foundation to determine all the effects that a call to **`m`** might cause.

In a multi-method all parameters that have an object type are used for the dispatching:

```
type
  Thing = ref object of RootObj
  Unit = ref object of Thing
    x: int

method collide(a, b: Thing) {.base, inline.} =
  quit "to override!"

method collide(a: Thing, b: Unit) {.inline.} =
  echo "1"

method collide(a: Unit, b: Thing) {.inline.} =
  echo "2"

var a, b: Unit
new a
new b
collide(a, b) # output: 2
```

Invocation of a multi-method cannot be ambiguous: collide 2 is preferred over collide 1 because the resolution works from left to right. In the example `Unit, Thing` is preferred over `Thing, Unit`.

Performance note: Nim does not produce a virtual method table, but generates dispatch trees. This avoids the expensive indirect branch for method calls and enables inlining. However, other optimizations like compile time evaluation or dead code elimination do not work with methods.

Iterators and the for statement

The for statement is an abstract mechanism to iterate over the elements of a container. It relies on an iterator to do so. Like **while** statements, **for** statements open an implicit block, so that they can be left with a **break** statement.

The **for** loop declares iteration variables - their scope reaches until the end of the loop body. The iteration variables' types are inferred by the return type of the iterator.

An iterator is similar to a procedure, except that it can be called in the context of a **for** loop. Iterators provide a way to specify the iteration over an abstract type. A key role in the execution of a **for** loop plays the **yield** statement in the called iterator. Whenever a **yield** statement is reached the data is bound to the **for** loop variables and control continues in the body of the **for** loop. The iterator's local variables and execution state are automatically saved between calls. Example:

```
# this definition exists in the system module
iterator items*(a: string): char {.inline.} =
  var i = 0
  while i < len(a):
    yield a[i]
    inc(i)

for ch in items("hello world"): # `ch` is an iteration variable
  echo ch
```

The compiler generates code as if the programmer would have written this:

```
var i = 0
while i < len(a):
  var ch = a[i]
  echo ch
  inc(i)
```

If the iterator yields a tuple, there can be as many iteration variables as there are components in the tuple. The i'th iteration variable's type is the type of the i'th component. In other words, implicit tuple unpacking in a for loop context is supported.

Implict items/pairs invocations

If the for loop expression **e** does not denote an iterator and the for loop has exactly 1 variable, the for loop expression is rewritten to **items(e)**; ie. an **items** iterator is implicitly invoked:

```
for x in [1,2,3]: echo x
```

If the for loop has exactly 2 variables, a **pairs** iterator is implicitly invoked.

Symbol lookup of the identifiers **items/pairs** is performed after the rewriting step, so that all overloads of **items/pairs** are taken into account.

First class iterators

There are 2 kinds of iterators in Nim: *inline* and *closure* iterators. An inline iterator is an iterator that's always inlined by the compiler leading to zero overhead for the abstraction, but may result in a heavy increase in code size. Inline iterators are second class citizens; They can be passed as parameters only to other inlining code facilities like templates, macros and other inline iterators.

In contrast to that, a closure iterator can be passed around more freely:

```
iterator count0(): int {.closure.} =
  yield 0

iterator count2(): int {.closure.} =
  var x = 1
  yield x
  inc x
  yield x

proc invoke(iter: iterator(): int {.closure.}) =
  for x in iter(): echo x

invoke(count0)
invoke(count2)
```

Closure iterators have other restrictions than inline iterators:

1. **`yield`** in a closure iterator can not occur in a **`try`** statement.
2. For now, a closure iterator cannot be evaluated at compile time.
3. **`return`** is allowed in a closure iterator (but rarely useful) and ends iteration.
4. Neither inline nor closure iterators can be recursive.

Iterators that are neither marked **`{.closure.}`** nor **`{.inline.}`** explicitly default to being inline, but this may change in future versions of the implementation.

The **`iterator`** type is always of the calling convention **`closure`** implicitly; the following example shows how to use iterators to implement a collaborative tasking system:

```
# simple tasking:
type
  Task = iterator (ticker: int)

iterator a1(ticker: int) {.closure.} =
  echo "a1: A"
  yield
  echo "a1: B"
  yield
  echo "a1: C"
  yield
  echo "a1: D"

iterator a2(ticker: int) {.closure.} =
  echo "a2: A"
  yield
  echo "a2: B"
  yield
  echo "a2: C"

proc runTasks(t: varargs[Task]) =
  var ticker = 0
  while true:
    let x = t[ticker mod t.len]
    if finished(x): break
    x(ticker)
    inc ticker

runTasks(a1, a2)
```

The builtin **`system.finished`** can be used to determine if an iterator has finished its operation;

no exception is raised on an attempt to invoke an iterator that has already finished its work.

Note that **system.finished** is error prone to use because it only returns **true** one iteration after the iterator has finished:

```
iterator mycount(a, b: int): int {.closure.} =
  var x = a
  while x <= b:
    yield x
    inc x

var c = mycount # instantiate the iterator
while not finished(c):
  echo c(1, 3)

# Produces
1
2
3
0
```

Instead this code has to be used:

```
var c = mycount # instantiate the iterator
while true:
  let value = c(1, 3)
  if finished(c): break # and discard 'value'!
  echo value
```

It helps to think that the iterator actually returns a pair **(value, done)** and **finished** is used to access the hidden **done** field.

Closure iterators are *resumable functions* and so one has to provide the arguments to every call. To get around this limitation one can capture parameters of an outer factory proc:

```
proc mycount(a, b: int): iterator (): int =
  result = iterator (): int =
    var x = a
    while x <= b:
      yield x
      inc x

let foo = mycount(1, 4)

for f in foo():
  echo f
```

Converters

A converter is like an ordinary proc except that it enhances the "implicitly convertible" type relation (see Convertible relation):

```
# bad style ahead: Nim is not C.
converter toBool(x: int): bool = x != 0

if 4:
  echo "compiles"
```

A converter can also be explicitly invoked for improved readability. Note that implicit converter chaining is not supported: If there is a converter from type A to type B and from type B to type C the implicit conversion from A to C is not provided.

Type sections

Example:

```
type # example demonstrating mutually recursive types
  Node = ref NodeObj # a traced pointer to a NodeObj
  NodeObj = object
    le, ri: Node     # left and right subtrees
    sym: ref Sym     # leaves contain a reference to a Sym

  Sym = object       # a symbol
    name: string     # the symbol's name
    line: int        # the line the symbol was declared in
    code: Node       # the symbol's abstract syntax tree
```

A type section begins with the **`type`** keyword. It contains multiple type definitions. A type definition binds a type to a name. Type definitions can be recursive or even mutually recursive. Mutually recursive types are only possible within a single **`type`** section. Nominal types like **`objects`** or **`enums`** can only be defined in a **`type`** section.

Exception handling

Try statement

Example:

```
# read the first two lines of a text file that should contain numbers
# and tries to add them
var
  f: File
if open(f, "numbers.txt"):
  try:
    var a = readLine(f)
    var b = readLine(f)
    echo "sum: " & $(parseInt(a) + parseInt(b))
  except OverflowError:
    echo "overflow!"
  except ValueError:
    echo "could not convert string to integer"
  except IOError:
    echo "IO error!"
  except:
    echo "Unknown exception!"
  finally:
    close(f)
```

The statements after the **try** are executed in sequential order unless an exception **e** is raised. If the exception type of **e** matches any listed in an **except** clause the corresponding statements are executed. The statements following the **except** clauses are called exception handlers.

The empty except clause is executed if there is an exception that is not listed otherwise. It is similar to an **else** clause in **if** statements.

If there is a finally clause, it is always executed after the exception handlers.

The exception is *consumed* in an exception handler. However, an exception handler may raise another exception. If the exception is not handled, it is propagated through the call stack. This means that often the rest of the procedure - that is not within a **finally** clause - is not executed (if an exception occurs).

Try expression

Try can also be used as an expression; the type of the **try** branch then needs to fit the types of **except** branches, but the type of the **finally** branch always has to be **void**:

```
let x = try: parseInt("133a")
        except: -1
        finally: echo "hi"
```

To prevent confusing code there is a parsing limitation; if the **try** follows a **(** it has to be written as a one liner:

```
let x = (try: parseInt("133a") except: -1)
```

Except clauses

Within an **except** clause, it is possible to use **getCurrentException** to retrieve the exception that has been raised:

```
try:
  # ...
except IOError:
  let e = getCurrentException()
  # Now use "e"
```

Note that **getCurrentException** always returns a **ref Exception** type. If a variable of the proper type is needed (in the example above, **IOError**), one must convert it explicitly:

```
try:
  # ...
except IOError:
  let e = (ref IOError)(getCurrentException())
  # "e" is now of the proper type
```

However, this is seldom needed. The most common case is to extract an error message from **e**, and for such situations it is enough to use **getCurrentExceptionMsg**:

```
try:
  # ...
except IOError:
  echo "I/O error: " & getCurrentExceptionMsg()
```

Defer statement

Instead of a **try finally** statement a **defer** statement can be used.

Any statements following the **defer** in the current block will be considered to be in an implicit try block:

```
var f = open("numbers.txt")
defer: close(f)
f.write "abc"
f.write "def"
```

Is rewritten to:

```
var f = open("numbers.txt")
try:
  f.write "abc"
  f.write "def"
finally:
  close(f)
```

Top level **defer** statements are not supported since it's unclear what such a statement should refer to.

Raise statement

Example:

```
raise newEOS("operating system failed")
```

Apart from built-in operations like array indexing, memory allocation, etc. the **raise** statement is the only way to raise an exception.

If no exception name is given, the current exception is re-raised. The ReraiseError exception is raised if there is no exception to re-raise. It follows that the **raise** statement *always* raises an exception.

Exception hierarchy

The exception tree is defined in the system (system.html) module:

- Exception (system.html#Exception)
  - AccessViolationError (system.html#AccessViolationError)
  - ArithmeticError (system.html#ArithmeticError)
    - DivByZeroError (system.html#DivByZeroError)
    - OverflowError (system.html#OverflowError)
  - AssertionError (system.html#AssertionError)
  - DeadThreadError (system.html#DeadThreadError)
  - FloatingPointError (system.html#FloatingPointError)
    - FloatDivByZeroError (system.html#FloatDivByZeroError)
    - FloatInexactError (system.html#FloatInexactError)
    - FloatInvalidOpError (system.html#FloatInvalidOpError)
    - FloatOverflowError (system.html#FloatOverflowError)
    - FloatUnderflowError (system.html#FloatUnderflowError)
  - FieldError (system.html#FieldError)
  - IndexError (system.html#IndexError)
  - ObjectAssignmentError (system.html#ObjectAssignmentError)
  - ObjectConversionError (system.html#ObjectConversionError)
  - ValueError (system.html#ValueError)
    - KeyError (system.html#KeyError)
  - ReraiseError (system.html#ReraiseError)
  - RangeError (system.html#RangeError)
  - OutOfMemoryError (system.html#OutOfMemoryError)
  - ResourceExhaustedError (system.html#ResourceExhaustedError)
  - StackOverflowError (system.html#StackOverflowError)
  - SystemError (system.html#SystemError)
    - IOError (system.html#IOError)
    - OSError (system.html#OSError)
      - LibraryError (system.html#LibraryError)

Effect system

Exception tracking

Nim supports exception tracking. The raises pragma can be used to explicitly define which exceptions a proc/iterator/method/converter is allowed to raise. The compiler verifies this:

```
proc p(what: bool) {.raises: [IOError, OSError].} =
  if what: raise newException(IOError, "IO")
  else: raise newException(OSError, "OS")
```

An empty **raises** list (**raises: []**) means that no exception may be raised:

```
proc p(): bool {.raises: [].} =
  try:
    unsafeCall()
    result = true
  except:
    result = false
```

A **raises** list can also be attached to a proc type. This affects type compatibility:

```
type
  Callback = proc (s: string) {.raises: [IOError].}
var
  c: Callback

proc p(x: string) =
  raise newException(OSError, "OS")

c = p # type error
```

For a routine **p** the compiler uses inference rules to determine the set of possibly raised exceptions; the algorithm operates on **p**'s call graph:

1. Every indirect call via some proc type **T** is assumed to raise **system.Exception** (the base type of the exception hierarchy) and thus any exception unless **T** has an explicit **raises** list. However if the call is of the form **f(...)** where **f** is a parameter of the currently analysed routine it is ignored. The call is optimistically assumed to have no effect. Rule 2 compensates for this case.
2. Every expression of some proc type within a call that is not a call itself (and not nil) is assumed to be called indirectly somehow and thus its raises list is added to **p**'s raises list.
3. Every call to a proc **q** which has an unknown body (due to a forward declaration or an **importc** pragma) is assumed to raise **system.Exception** unless **q** has an explicit **raises** list.
4. Every call to a method **m** is assumed to raise **system.Exception** unless **m** has an explicit **raises** list.
5. For every other call the analysis can determine an exact **raises** list.
6. For determining a **raises** list, the **raise** and **try** statements of **p** are taken into consideration.

Rules 1-2 ensure the following works:

```
proc noRaise(x: proc()) {.raises: [].} =
  # unknown call that might raise anything, but valid:
  x()

proc doRaise() {.raises: [IOError].} =
  raise newException(IOError, "IO")

proc use() {.raises: [].} =
  # doesn't compile! Can raise IOError!
  noRaise(doRaise)
```

So in many cases a callback does not cause the compiler to be overly conservative in its effect analysis.

Tag tracking

The exception tracking is part of Nim's effect system. Raising an exception is an *effect*. Other effects can also be defined. A user defined effect is a means to *tag* a routine and to perform checks against this tag:

```
type IO = object ## input/output effect
proc readLine(): string {.tags: [IO].}

proc no_IO_please() {.tags: [].} =
  # the compiler prevents this:
  let x = readLine()
```

A tag has to be a type name. A **`tags`** list - like a **`raises`** list - can also be attached to a proc type. This affects type compatibility.

The inference for tag tracking is analogous to the inference for exception tracking.

Read/Write tracking

Note: Read/write tracking is not yet implemented!

The inference for read/write tracking is analogous to the inference for exception tracking.

Effects pragma

The **`effects`** pragma has been designed to assist the programmer with the effects analysis. It is a statement that makes the compiler output all inferred effects up to the **`effects`**'s position:

```
proc p(what: bool) =
  if what:
    raise newException(IOError, "IO")
    {.effects.}
  else:
    raise newException(OSError, "OS")
```

The compiler produces a hint message that **`IOError`** can be raised. **`OSError`** is not listed as it cannot be raised in the branch the **`effects`** pragma appears in.

Generics

Generics are Nim's means to parametrize procs, iterators or types with type parameters. Depending on context, the brackets are used either to introduce type parameters or to instantiate a generic proc, iterator or type.

The following example shows a generic binary tree can be modelled:

```
type
  BinaryTreeObj[T] = object    # BinaryTreeObj is a generic type with
                               # with generic param ``T``
    le, ri: BinaryTree[T]      # left and right subtrees; may be nil
    data: T                    # the data stored in a node
  BinaryTree[T] = ref BinaryTreeObj[T] # a shorthand for notational convenience

proc newNode[T](data: T): BinaryTree[T] = # constructor for a node
  new(result)
  result.data = data

proc add[T](root: var BinaryTree[T], n: BinaryTree[T]) =
  if root == nil:
    root = n
  else:
    var it = root
    while it != nil:
      var c = cmp(it.data, n.data) # compare the data items; uses
                                   # the generic ``cmp`` proc that works for
                                   # any type that has a ``==`` and ``<``
                                   # operator
      if c < 0:
        if it.le == nil:
          it.le = n
          return
        it = it.le
      else:
        if it.ri == nil:
          it.ri = n
          return
        it = it.ri

iterator inorder[T](root: BinaryTree[T]): T =
  # inorder traversal of a binary tree
  # recursive iterators are not yet implemented, so this does not work in
  # the current compiler!
  if root.le != nil: yield inorder(root.le)
  yield root.data
  if root.ri != nil: yield inorder(root.ri)

var
  root: BinaryTree[string]  # instantiate a BinaryTree with the type string
add(root, newNode("hallo")) # instantiates generic procs ``newNode`` and
add(root, newNode("world")) # ``add``
for str in inorder(root):
  writeLine(stdout, str)
```

Is operator

The **is** operator checks for type equivalence at compile time. It is therefore very useful for type specialization within generic code:

```
type
  Table[Key, Value] = object
    keys: seq[Key]
    values: seq[Value]
    when not (Key is string): # nil value for strings used for optimization
      deletedKeys: seq[bool]
```

Type operator

The **type** (in many other languages called typeof) operator can be used to get the type of an expression:

```
var x = 0
var y: type(x) # y has type int
```

If **type** is used to determine the result type of a proc/iterator/converter call **c(X)** (where **X** stands for a possibly empty list of arguments), the interpretation where **c** is an iterator is preferred over the other interpretations:

```
import strutils

# strutils contains both a ``split`` proc and iterator, but since an
# an iterator is the preferred interpretation, `y` has the type ``string``:
var y: type("a b c".split)
```

Type Classes

A type class is a special pseudo-type that can be used to match against types in the context of overload resolution or the **is** operator. Nim supports the following built-in type classes:

type class	matches
`object`	any object type
`tuple`	any tuple type
`enum`	any enumeration
`proc`	any proc type
`ref`	any **ref** type
`ptr`	any **ptr** type
`var`	any **var** type
`distinct`	any distinct type
`array`	any array type
`set`	any set type
`seq`	any seq type
`auto`	any type
`any`	distinct auto (see below)

Furthermore, every generic type automatically creates a type class of the same name that will match any instantiation of the generic type.

Type classes can be combined using the standard boolean operators to form more complex type classes:

```
# create a type class that will match all tuple and object types
type RecordType = tuple or object

proc printFields(rec: RecordType) =
  for key, value in fieldPairs(rec):
    echo key, " = ", value
```

Procedures utilizing type classes in such manner are considered to be implicitly generic. They will be instantiated once for each unique combination of param types used within the program.

Nim also allows for type classes and regular types to be specified as type constraints of the generic type parameter:

```
proc onlyIntOrString[T: int|string](x, y: T) = discard

onlyIntOrString(450, 616) # valid
onlyIntOrString(5.0, 0.0) # type mismatch
onlyIntOrString("xy", 50) # invalid as 'T' cannot be both at the same time
```

By default, during overload resolution each named type class will bind to exactly one concrete type. We call such type classes bind once types. Here is an example taken directly from the system module to illustrate this:

```
proc `==`*(x, y: tuple): bool =
  ## requires `x` and `y` to be of the same tuple type
  ## generic ``==`` operator for tuples that is lifted from the components
  ## of `x` and `y`.
  result = true
  for a, b in fields(x, y):
    if a != b: result = false
```

Alternatively, the **`distinct`** type modifier can be applied to the type class to allow each param matching the type class to bind to a different type. Such type classes are called bind many types.

Procs written with the implicitly generic style will often need to refer to the type parameters of the matched generic type. They can be easily accessed using the dot syntax:

```
type Matrix[T, Rows, Columns] = object
  ...

proc `[]`(m: Matrix, row, col: int): Matrix.T =
  m.data[col * high(Matrix.Columns) + row]
```

Alternatively, the *type* operator can be used over the proc params for similar effect when anonymous or distinct type classes are used.

When a generic type is instantiated with a type class instead of a concrete type, this results in another more specific type class:

```
seq[ref object]  # Any sequence storing references to any object type

type T1 = auto
proc foo(s: seq[T1], e: T1)
  # seq[T1] is the same as just `seq`, but T1 will be allowed to bind
  # to a single type, while the signature is being matched

Matrix[Ordinal] # Any Matrix instantiation using integer values
```

As seen in the previous example, in such instantiations, it's not necessary to supply all type parameters of the generic type, because any missing ones will be inferred to have the equivalent of the *any* type class and thus they will match anything without discrimination.

Concepts

Note: Concepts are still in development.

Concepts, also known as "user-defined type classes", are used to specify an arbitrary set of requirements that the matched type must satisfy.

Concepts are written in the following form:

```
type
  Comparable = concept x, y
    (x < y) is bool

  Stack[T] = concept s, var v
    s.pop() is T
    v.push(T)

    s.len is Ordinal

    for value in s:
      value is T
```

The concept is a match if:

1. all of the expressions within the body can be compiled for the tested type
2. all statically evaluable boolean expressions in the body must be true

The identifiers following the `concept` keyword represent instances of the currently matched type. You can apply any of the standard type modifiers such as `var`, `ref`, `ptr` and `static` to denote a more specific type of instance. You can also apply the *type* modifier to create a named instance of the type itself:

```
type
  MyConcept = concept x, var v, ref r, ptr p, static s, type T
    ...
```

Within the concept body, types can appear in positions where ordinary values and parameters are expected. This provides a more convenient way to check for the presence of callable symbols with specific signatures:

```
type
  OutputStream = concept var s
    s.write(string)
```

In order to check for symbols accepting `typedesc` params, you must prefix the type with an explicit `type` modifier. The named instance of the type, following the `concept` keyword is also considered an explicit `typedesc` value that will be matched only as a type.

```
type
  # Let's imagine a user-defined casting framework with operators
  # such as `val.to(string)` and `val.to(JSonValue)`. We can test
  # for these with the following concept:
  MyCastables = concept x
    x.to(type string)
    x.to(type JSonValue)

  # Let's define a couple of concepts, known from Algebra:
  AdditiveMonoid* = concept x, y, type T
    x + y is T
    T.zero is T # require a proc such as `int.zero` or 'Position.zero'

  AdditiveGroup* = concept x, y, type T
    x is AdditiveMonoid
    -x is T
    x - y is T
```

Please note that the **is** operator allows one to easily verify the precise type signatures of the required operations, but since type inference and default parameters are still applied in the concept body, it's also possible to describe usage protocols that do not reveal implementation details.

Much like generics, concepts are instantiated exactly once for each tested type and any static code included within the body is executed only once.

Concept diagnostics

By default, the compiler will report the matching errors in concepts only when no other overload can be selected and a normal compilation error is produced. When you need to understand why the compiler is not matching a particular concept and, as a result, a wrong overload is selected, you can apply the **explain** pragma to either the concept body or a particular call-site.

```
type
  MyConcept {.explain.} = concept ...

overloadedProc(x, y, z) {.explain.}
```

This will provide Hints in the compiler output either every time the concept is not matched or only on the particular call-site.

Generic concepts and type binding rules

The concept types can be parametric just like the regular generic types:

```
### matrixalgo.nim

import typetraits

type
  AnyMatrix*[R, C: static[int]; T] = concept m, var mvar, type M
    M.ValueType is T
    M.Rows == R
    M.Cols == C

    m[int, int] is T
    mvar[int, int] = T

    type TransposedType = stripGenericParams(M)[C, R, T]

  AnySquareMatrix*[N: static[int], T] = AnyMatrix[N, N, T]

  AnyTransform3D* = AnyMatrix[4, 4, float]

proc transposed*(m: AnyMatrix): m.TransposedType =
  for r in 0 .. <m.R:
    for c in 0 .. <m.C:
      result[r, c] = m[c, r]

proc determinant*(m: AnySquareMatrix): int =
  ...

proc setPerspectiveProjection*(m: AnyTransform3D) =
  ...

--------------
### matrix.nim

type
  Matrix*[M, N: static[int]; T] = object
    data: array[M*N, T]

proc `[]`*(M: Matrix; m, n: int): M.T =
  M.data[m * M.N + n]

proc `[]=`*(M: var Matrix; m, n: int; v: M.T) =
  M.data[m * M.N + n] = v

# Adapt the Matrix type to the concept's requirements
template Rows*(M: type Matrix): expr = M.M
template Cols*(M: type Matrix): expr = M.N
template ValueType*(M: type Matrix): typedesc = M.T

-------------
### usage.nim

import matrix, matrixalgo

var
  m: Matrix[3, 3, int]
  projectionMatrix: Matrix[4, 4, float]

echo m.transposed.determinant
```

When the concept type is matched against a concrete type, the unbound type parameters are inferred from the body of the concept in a way that closely resembles the way generic parameters of callable symbols are inferred on call sites.

Unbound types can appear both as params to calls such as *s.push(T)* and on the right-hand side of the **is** operator in cases such as *x.pop is T* and *x.data is seq[T]*.

Unbound static params will be inferred from expressions involving the `==` operator and also when types dependent on them are being matched:

```
type
  MatrixReducer[M, N: static[int]; T] = concept x
    x.reduce(SquareMatrix[N, T]) is array[M, int]
```

The Nim compiler includes a simple linear equation solver, allowing it to infer static params in some situations where integer arithmetic is involved.

Just like in regular type classes, Nim discriminates between **bind once** and **bind many** types when matching the concept. You can add the **distinct** modifier to any of the otherwise inferable types to get a type that will be matched without permanently inferring it. This may be useful when you need to match several procs accepting the same wide class of types:

```
type
  Enumerable[T] = concept e
    for v in e:
      v is T

type
  MyConcept = concept o
    # this could be inferred to a type such as Enumerable[int]
    o.foo is distinct Enumerable

    # this could be inferred to a different type such as Enumerable[float]
    o.bar is distinct Enumerable

    # it's also possible to give an alias name to a `bind many` type class
    type Enum = distinct Enumerable
    o.baz is Enum
```

On the other hand, using **bind once** types allows you to test for equivalent types used in multiple signatures, without actually requiring any concrete types, thus allowing you to encode implementation-defined types:

```
type
  MyConcept = concept x
    type T1 = auto
    x.foo(T1)
    x.bar(T1) # both procs must accept the same type

    type T2 = seq[SomeNumber]
    x.alpha(T2)
    x.omega(T2) # both procs must accept the same type
                # and it must be a numeric sequence
```

As seen in the previous examples, you can refer to generic concepts such as *Enumerable[T]* just by their short name. Much like the regular generic types, the concept will be automatically instantiated with the bind once auto type in the place of each missing generic param.

Please note that generic concepts such as *Enumerable[T]* can be matched against concrete types such as *string*. Nim doesn't require the concept type to have the same number of parameters as the type being matched. If you wish to express a requirement towards the generic parameters of the matched type, you can use a type mapping operator such as *genericHead* or *stripGenericParams* within the body of the concept to obtain the uninstantiated version of the type, which you can then try to instantiate in any required way. For example, here is how one might define the classic *Functor* concept from Haskell and then demonstrate that Nim's *Option[T]* type is an instance of it:

```
import future, typetraits

type
  Functor[A] = concept f
    type MatchedGenericType = genericHead(f.type)
      # `f` will be a value of a type such as `Option[T]`
      # `MatchedGenericType` will become the `Option` type

    f.val is A
      # The Functor should provide a way to obtain
      # a value stored inside it

    type T = auto
    map(f, A -> T) is MatchedGenericType[T]
      # And it should provide a way to map one instance of
      # the Functor to a instance of a different type, given
      # a suitable `map` operation for the enclosed values

import options
echo Option[int] is Functor # prints true
```

Concept derived values

All top level constants or types appearing within the concept body are accessible through the dot operator in procs where the concept was successfully matched to a concrete type:

```
type
  DateTime = concept t1, t2, type T
    const Min = T.MinDate
    T.Now is T

    t1 < t2 is bool

    type TimeSpan = type(t1 - t2)
    TimeSpan * int is TimeSpan
    TimeSpan + TimeSpan is TimeSpan

    t1 + TimeSpan is T

proc eventsJitter(events: Enumerable[DateTime]): float =
  var
    # this variable will have the inferred TimeSpan type for
    # the concrete Date-like value the proc was called with:
    averageInterval: DateTime.TimeSpan

    deviation: float
  ...
```

Concept refinement

When the matched type within a concept is directly tested against a different concept, we say that the outer concept is a refinement of the inner concept and thus it is more-specific. When both concepts are matched in a call during overload resolution, Nim will assign a higher precedence to the most specific one. As an alternative way of defining concept refinements, you can use the object inheritance syntax involving the **of** keyword:

```
type
  Graph = concept g, type G of EqualyComparable, Copyable
    type
      VertexType = G.VertexType
      EdgeType = G.EdgeType

    VertexType is Copyable
    EdgeType is Copyable

    var
      v: VertexType
      e: EdgeType

  IncidendeGraph = concept of Graph
    # symbols such as variables and types from the refined
    # concept are automatically in scope:

    g.source(e) is VertexType
    g.target(e) is VertexType

    g.outgoingEdges(v) is Enumerable[EdgeType]

  BidirectionalGraph = concept g, type G
    # The following will also turn the concept into a refinement when it
    # comes to overload resolution, but it doesn't provide the convenient
    # symbol inheritance
    g is IncidendeGraph

    g.incomingEdges(G.VertexType) is Enumerable[G.EdgeType]

proc f(g: IncidendeGraph)
proc f(g: BidirectionalGraph) # this one will be preferred if we pass a type
                              # matching the BidirectionalGraph concept
```

Converter type classes

Concepts can also be used to convert a whole range of types to a single type or a small set of simpler types. This is achieved with a *return* statement within the concept body:

```
type
  Stringable = concept x
    $x is string
    return $x

  StringRefValue[CharType] = object
    base: ptr CharType
    len: int

  StringRef = concept x
    # the following would be an overloaded proc for cstring, string, seq and
    # other user-defined types, returning either a StringRefValue[char] or
    # StringRefValue[wchar]
    return makeStringRefValue(x)

# the varargs param will here be converted to an array of StringRefValues
# the proc will have only two instantiations for the two character types
proc log(format: static[string], varargs[StringRef])

# this proc will allow char and wchar values to be mixed in
# the same call at the cost of additional instantiations
# the varargs param will be converted to a tuple
proc log(format: static[string], varargs[distinct StringRef])
```

VTable types

Concepts allow Nim to define a great number of algorithms, using only static polymorphism and without erasing any type information or sacrificing any execution speed. But when polymorphic collections of objects are required, the user must use one of the provided type erasure techniques - either common base types or VTable types.

VTable types are represented as "fat pointers" storing a reference to an object together with a reference to a table of procs implementing a set of required operations (the so called vtable).

In contrast to other programming languages, the vtable in Nim is stored externally to the object, allowing you to create multiple different vtable views for the same object. Thus, the polymorphism in Nim is unbounded - any type can implement an unlimited number of protocols or interfaces not originally envisioned by the type's author.

Any concept type can be turned into a VTable type by using the `vtref` or the `vtptr` compiler magics. Under the hood, these magics generate a converter type class, which converts the regular instances of the matching types to the corresponding VTable type.

```
type
  IntEnumerable = vtref Enumerable[int]

  MyObject = object
    enumerables: seq[IntEnumerable]
    streams: seq[OutputStream.vtref]

proc addEnumerable(o: var MyObject, e: IntEnumerable) =
  o.enumerables.add e

proc addStream(o: var MyObject, e: OutputStream.vtref) =
  o.streams.add e
```

The procs that will be included in the vtable are derived from the concept body and include all proc calls for which all param types were specified as concrete types. All such calls should include exactly one param of the type matched against the concept (not necessarily in the first position), which will be considered the value bound to the vtable.

Overloads will be created for all captured procs, accepting the vtable type in the position of the captured underlying object.

Under these rules, it's possible to obtain a vtable type for a concept with unbound type parameters or one instantiated with metatypes (type classes), but it will include a smaller number of captured procs. A completely empty vtable will be reported as an error.

The `vtref` magic produces types which can be bound to `ref` types and the `vtptr` magic produced types bound to `ptr` types.

Symbol lookup in generics

The symbol binding rules in generics are slightly subtle: There are "open" and "closed" symbols. A "closed" symbol cannot be re-bound in the instantiation context, an "open" symbol can. Per default overloaded symbols are open and every other symbol is closed.

Open symbols are looked up in two different contexts: Both the context at definition and the context at instantiation are considered:

```
type
  Index = distinct int

proc `==` (a, b: Index): bool {.borrow.}

var a = (0, 0.Index)
var b = (0, 0.Index)

echo a == b # works!
```

In the example the generic `==` for tuples (as defined in the system module) uses the `==` operators of the tuple's components. However, the `==` for the `Index` type is defined *after* the `==` for tuples; yet the example compiles as the instantiation takes the currently defined symbols into account too.

A symbol can be forced to be open by a mixin declaration:

```
proc create*[T](): ref T =
  # there is no overloaded 'init' here, so we need to state that it's an
  # open symbol explicitly:
  mixin init
  new result
  init result
```

Bind statement

The `bind` statement is the counterpart to the `mixin` statement. It can be used to explicitly declare identifiers that should be bound early (i.e. the identifiers should be looked up in the scope of the template/generic definition):

```
# Module A
var
  lastId = 0

template genId*: untyped =
  bind lastId
  inc(lastId)
  lastId
```

```
# Module B
import A

echo genId()
```

But a **`bind`** is rarely useful because symbol binding from the definition scope is the default.

Templates

A template is a simple form of a macro: It is a simple substitution mechanism that operates on Nim's abstract syntax trees. It is processed in the semantic pass of the compiler.

The syntax to *invoke* a template is the same as calling a procedure.

Example:

```
template `!=` (a, b: untyped): untyped =
  # this definition exists in the System module
  not (a == b)

assert(5 != 6) # the compiler rewrites that to: assert(not (5 == 6))
```

The **`!=`**, **`>`**, **`>=`**, **`in`**, **`notin`**, **`isnot`** operators are in fact templates:

`a > b` is transformed into **`b < a`**.
`a in b` is transformed into **`contains(b, a)`**.
`notin` and **`isnot`** have the obvious meanings.

The "types" of templates can be the symbols **`untyped`**, **`typed`** or **`typedesc`** (stands for *type description*). These are "meta types", they can only be used in certain contexts. Real types can be used too; this implies that **`typed`** expressions are expected.

Typed vs untyped parameters

An **`untyped`** parameter means that symbol lookups and type resolution is not performed before the expression is passed to the template. This means that for example *undeclared* identifiers can be passed to the template:

```
template declareInt(x: untyped) =
  var x: int

declareInt(x) # valid
x = 3
```

```
template declareInt(x: typed) =
  var x: int

declareInt(x) # invalid, because x has not been declared and so has no type
```

A template where every parameter is **`untyped`** is called an immediate template. For historical reasons templates can be explicitly annotated with an **`immediate`** pragma and then these templates do not take part in overloading resolution and the parameters' types are *ignored* by the compiler. Explicit immediate templates are now deprecated.

Note: For historical reasons **`stmt`** is an alias for **`typed`** and **`expr`** an alias for **`untyped`**, but new code should use the newer, clearer names.

Passing a code block to a template

You can pass a block of statements as a last parameter to a template via a special **`:`** syntax:

```
template withFile(f, fn, mode, actions: untyped): untyped =
  var f: File
  if open(f, fn, mode):
    try:
      actions
    finally:
      close(f)
  else:
    quit("cannot open: " & fn)

withFile(txt, "ttempl3.txt", fmWrite):
  txt.writeLine("line 1")
  txt.writeLine("line 2")
```

In the example the two **writeLine** statements are bound to the **actions** parameter.

Usually to pass a block of code to a template the parameter that accepts the block needs to be of type **untyped**. Because symbol lookups are then delayed until template instantiation time:

```
template t(body: typed) =
  block:
    body

t:
  var i = 1
  echo i

t:
  var i = 2  # fails with 'attempt to redeclare i'
  echo i
```

The above code fails with the mysterious error message that **i** has already been declared. The reason for this is that the **var i = ...** bodies need to be type-checked before they are passed to the **body** parameter and type checking in Nim implies symbol lookups. For the symbol lookups to succeed **i** needs to be added to the current (i.e. outer) scope. After type checking these additions to the symbol table are not rolled back (for better or worse). The same code works with **untyped** as the passed body is not required to be type-checked:

```
template t(body: untyped) =
  block:
    body

t:
  var i = 1
  echo i

t:
  var i = 2  # compiles
  echo i
```

Varargs of untyped

In addition to the **untyped** meta-type that prevents type checking there is also **varargs[untyped]** so that not even the number of parameters is fixed:

```
template hideIdentifiers(x: varargs[untyped]) = discard

hideIdentifiers(undeclared1, undeclared2)
```

However, since a template cannot iterate over varargs, this feature is generally much more useful for macros.

Note: For historical reasons `varargs[expr]` is not equivalent to `varargs[untyped]`.

Symbol binding in templates

A template is a hygienic macro and so opens a new scope. Most symbols are bound from the definition scope of the template:

```
# Module A
var
  lastId = 0

template genId*: untyped =
  inc(lastId)
  lastId
```

```
# Module B
import A

echo genId() # Works as 'lastId' has been bound in 'genId's defining scope
```

As in generics symbol binding can be influenced via `mixin` or `bind` statements.

Identifier construction

In templates identifiers can be constructed with the backticks notation:

```
template typedef(name: untyped, typ: typedesc) =
  type
    `T name`* {.inject.} = typ
    `P name`* {.inject.} = ref `T name`

typedef(myint, int)
var x: PMyInt
```

In the example `name` is instantiated with `myint`, so \`T name\` becomes `Tmyint`.

Lookup rules for template parameters

A parameter `p` in a template is even substituted in the expression `x.p`. Thus template arguments can be used as field names and a global symbol can be shadowed by the same argument name even when fully qualified:

```
# module 'm'

type
  Lev = enum
    levA, levB

var abclev = levB

template tstLev(abclev: Lev) =
  echo abclev, " ", m.abclev

tstLev(levA)
# produces: 'levA levA'
```

But the global symbol can properly be captured by a **bind** statement:

```
# module 'm'

type
  Lev = enum
    levA, levB

var abclev = levB

template tstLev(abclev: Lev) =
  bind m.abclev
  echo abclev, " ", m.abclev

tstLev(levA)
# produces: 'levA levB'
```

Hygiene in templates

Per default templates are hygienic: Local identifiers declared in a template cannot be accessed in the instantiation context:

```
template newException*(exceptn: typedesc, message: string): untyped =
  var
    e: ref exceptn  # e is implicitly gensym'ed here
  new(e)
  e.msg = message
  e

# so this works:
let e = "message"
raise newException(EIO, e)
```

Whether a symbol that is declared in a template is exposed to the instantiation scope is controlled by the inject and gensym pragmas: gensym'ed symbols are not exposed but inject'ed are.

The default for symbols of entity **type**, **var**, **let** and **const** is **gensym** and for **proc**, **iterator**, **converter**, **template**, **macro** is **inject**. However, if the name of the entity is passed as a template parameter, it is an inject'ed symbol:

```
template withFile(f, fn, mode: untyped, actions: untyped): untyped =
  block:
    var f: File  # since 'f' is a template param, it's injected implicitly
    ...

withFile(txt, "ttempl3.txt", fmWrite):
  txt.writeLine("line 1")
  txt.writeLine("line 2")
```

The **inject** and **gensym** pragmas are second class annotations; they have no semantics outside of a template definition and cannot be abstracted over:

```
{.pragma myInject: inject.}

template t() =
  var x {.myInject.}: int # does NOT work
```

To get rid of hygiene in templates, one can use the dirty pragma for a template. **inject** and **gensym** have no effect in **dirty** templates.

Limitations of the method call syntax

The expression **x** in **x.f** needs to be semantically checked (that means symbol lookup and type checking) before it can be decided that it needs to be rewritten to **f(x)**. Therefore the dot syntax has some limiations when it is used to invoke templates/macros:

```
template declareVar(name: untyped) =
  const name {.inject.} = 45

# Doesn't compile:
unknownIdentifier.declareVar
```

Another common example is this:

```
from sequtils import toSeq

iterator something: string =
  yield "Hello"
  yield "World"

var info = toSeq(something())
```

The problem here is that the compiler already decided that **something()** as an iterator is not callable in this context before **toSeq** gets its chance to convert it into a sequence.

Macros

A macro is a special kind of low level template. Macros can be used to implement domain specific languages.

While macros enable advanced compile-time code transformations, they cannot change Nim's syntax. However, this is no real restriction because Nim's syntax is flexible enough anyway.

To write macros, one needs to know how the Nim concrete syntax is converted to an abstract syntax tree.

There are two ways to invoke a macro:

1. invoking a macro like a procedure call (*expression macros*)
2. invoking a macro with the special **`macrostmt`** syntax (*statement macros*)

Expression Macros

The following example implements a powerful **`debug`** command that accepts a variable number of arguments:

```
# to work with Nim syntax trees, we need an API that is defined in the
# ``macros`` module:
import macros

macro debug(n: varargs[untyped]): untyped =
  # `n` is a Nim AST that contains the whole macro invocation
  # this macro returns a list of statements:
  result = newNimNode(nnkStmtList, n)
  # iterate over any argument that is passed to this macro:
  for i in 0..n.len-1:
    # add a call to the statement list that writes the expression;
    # `toStrLit` converts an AST to its string representation:
    add(result, newCall("write", newIdentNode("stdout"), toStrLit(n[i])))
    # add a call to the statement list that writes ": "
    add(result, newCall("write", newIdentNode("stdout"), newStrLitNode(": ")))
    # add a call to the statement list that writes the expressions value:
    add(result, newCall("writeLine", newIdentNode("stdout"), n[i]))

var
  a: array [0..10, int]
  x = "some string"
a[0] = 42
a[1] = 45

debug(a[0], a[1], x)
```

The macro call expands to:

```
write(stdout, "a[0]")
write(stdout, ": ")
writeLine(stdout, a[0])

write(stdout, "a[1]")
write(stdout, ": ")
writeLine(stdout, a[1])

write(stdout, "x")
write(stdout, ": ")
writeLine(stdout, x)
```

Arguments that are passed to a **varargs** parameter are wrapped in an array constructor expression. This is why **debug** iterates over all of **n**'s children.

BindSym

The above **debug** macro relies on the fact that **write**, **writeLine** and **stdout** are declared in the system module and thus visible in the instantiating context. There is a way to use bound identifiers (aka symbols) instead of using unbound identifiers. The **bindSym** builtin can be used for that:

```
import macros

macro debug(n: varargs[typed]): untyped =
  result = newNimNode(nnkStmtList, n)
  for x in n:
    # we can bind symbols in scope via 'bindSym':
    add(result, newCall(bindSym"write", bindSym"stdout", toStrLit(x)))
    add(result, newCall(bindSym"write", bindSym"stdout", newStrLitNode(": ")))
    add(result, newCall(bindSym"writeLine", bindSym"stdout", x))

var
  a: array [0..10, int]
  x = "some string"
a[0] = 42
a[1] = 45

debug(a[0], a[1], x)
```

The macro call expands to:

```
write(stdout, "a[0]")
write(stdout, ": ")
writeLine(stdout, a[0])

write(stdout, "a[1]")
write(stdout, ": ")
writeLine(stdout, a[1])

write(stdout, "x")
write(stdout, ": ")
writeLine(stdout, x)
```

However, the symbols **write**, **writeLine** and **stdout** are already bound and are not looked up again. As the example shows, **bindSym** does work with overloaded symbols implicitly.

Statement Macros

Statement macros are defined just as expression macros. However, they are invoked by an expression following a colon.

The following example outlines a macro that generates a lexical analyzer from regular expressions:

```
import macros

macro case_token(n: untyped): untyped =
  # creates a lexical analyzer from regular expressions
  # ... (implementation is an exercise for the reader :-)
  discard

case_token: # this colon tells the parser it is a macro statement
of r"[A-Za-z_]+[A-Za-z_0-9]*":
  return tkIdentifier
of r"0-9+":
  return tkInteger
of r"[\+\-\*\?]+":
  return tkOperator
else:
  return tkUnknown
```

Style note: For code readability, it is the best idea to use the least powerful programming construct that still suffices. So the "check list" is:

1. Use an ordinary proc/iterator, if possible.
2. Else: Use a generic proc/iterator, if possible.
3. Else: Use a template, if possible.
4. Else: Use a macro.

Macros as pragmas

Whole routines (procs, iterators etc.) can also be passed to a template or a macro via the pragma notation:

```
template m(s: untyped) = discard

proc p() {.m.} = discard
```

This is a simple syntactic transformation into:

```
template m(s: untyped) = discard

m:
  proc p() = discard
```

Special Types

static[T]

Note: static[T] is still in development.

As their name suggests, static parameters must be known at compile-time:

```
proc precompiledRegex(pattern: static[string]): RegEx =
  var res {.global.} = re(pattern)
  return res

precompiledRegex("/d+") # Replaces the call with a precompiled
                        # regex, stored in a global variable

precompiledRegex(paramStr(1)) # Error, command-line options
                              # are not known at compile-time
```

For the purposes of code generation, all static params are treated as generic params - the proc will be compiled separately for each unique supplied value (or combination of values).

Static params can also appear in the signatures of generic types:

```
type
  Matrix[M,N: static[int]; T: Number] = array[0..(M*N - 1), T]
    # Note how `Number` is just a type constraint here, while
    # `static[int]` requires us to supply a compile-time int value

  AffineTransform2D[T] = Matrix[3, 3, T]
  AffineTransform3D[T] = Matrix[4, 4, T]

var m1: AffineTransform3D[float]  # OK
var m2: AffineTransform2D[string] # Error, `string` is not a `Number`
```

typedesc

typedesc is a special type allowing one to treat types as compile-time values (i.e. if types are compile-time values and all values have a type, then typedesc must be their type).

When used as a regular proc param, typedesc acts as a type class. The proc will be instantiated for each unique type parameter and one can refer to the instantiation type using the param name:

```
proc new(T: typedesc): ref T =
  echo "allocating ", T.name
  new(result)

var n = Node.new
var tree = new(BinaryTree[int])
```

When multiple typedesc params are present, they act like a distinct type class (i.e. they will bind freely to different types). To force a bind-once behavior one can use a named alias or an explicit *typedesc* generic param:

```
proc acceptOnlyTypePairs[T: typedesc, U: typedesc](A, B: T; C, D: U)
```

Once bound, typedesc params can appear in the rest of the proc signature:

```
template declareVariableWithType(T: typedesc, value: T) =
  var x: T = value

declareVariableWithType int, 42
```

Overload resolution can be further influenced by constraining the set of types that will match the typedesc param:

```
template maxval(T: typedesc[int]): int = high(int)
template maxval(T: typedesc[float]): float = Inf

var i = int.maxval
var f = float.maxval
var s = string.maxval # error, maxval is not implemented for string
```

The constraint can be a concrete type or a type class.

Special Operators

dot operators

Note: Dot operators are still experimental and so need to be enabled via `{.experimental.}`.

Nim offers a special family of dot operators that can be used to intercept and rewrite proc call and field access attempts, referring to previously undeclared symbol names. They can be used to provide a fluent interface to objects lying outside the static confines of the type system such as values from dynamic scripting languages or dynamic file formats such as JSON or XML.

When Nim encounters an expression that cannot be resolved by the standard overload resolution rules, the current scope will be searched for a dot operator that can be matched against a re-written form of the expression, where the unknown field or proc name is converted to an additional static string parameter:

```
a.b # becomes `.`(a, "b")
a.b(c, d) # becomes `.`(a, "b", c, d)
```

The matched dot operators can be symbols of any callable kind (procs, templates and macros), depending on the desired effect:

```
proc `.` (js: PJsonNode, field: string): JSON = js[field]

var js = parseJson("{ x: 1, y: 2}")
echo js.x # outputs 1
echo js.y # outputs 2
```

The following dot operators are available:

operator *.*

This operator will be matched against both field accesses and method calls.

operator *.()*

This operator will be matched exclusively against method calls. It has higher precedence than the . operator and this allows one to handle expressions like *x.y* and *x.y()* differently if one is interfacing with a scripting language for example.

operator *.=*

This operator will be matched against assignments to missing fields.

```
a.b = c # becomes `.=`(a, "b", c)
```

Type bound operations

There are 3 operations that are bound to a type:

1. Assignment
2. Destruction
3. Deep copying for communication between threads

These operations can be *overriden* instead of *overloaded*. This means the implementation is automatically lifted to structured types. For instance if type **`T`** has an overriden assignment operator **`=`** this operator is also used for assignments of the type **`seq[T]`**. Since these operations are bound to a type they have to be bound to a nominal type for reasons of simplicity of implementation: This means an overriden **`deepCopy`** for **`ref T`** is really bound to **`T`** and not to **`ref T`**. This also means that one cannot override **`deepCopy`** for both **`ptr T`** and **`ref T`** at the same time; instead a helper distinct or object type has to be used for one pointer type.

operator =

This operator is the assignment operator. Note that in the contexts **`result = expr`**, **`parameter = defaultValue`** or for parameter passing no assignment is performed. For a type **`T`** that has an overloaded assignment operator **`var v = T()`** is rewritten to **`var v: T; v = T()`**; in other words **`var`** and **`let`** contexts do count as assignments.

The assignment operator needs to be attached to an object or distinct type **`T`**. Its signature has to be **`(var T, T)`**. Example:

```
type
  Concrete = object
    a, b: string

proc `=`(d: var Concrete; src: Concrete) =
  shallowCopy(d.a, src.a)
  shallowCopy(d.b, src.b)
  echo "Concrete '=' called"

var x, y: array[0..2, Concrete]
var cA, cB: Concrete

var cATup, cBTup: tuple[x: int, ha: Concrete]

x = y
cA = cB
cATup = cBTup
```

destructors

A destructor must have a single parameter with a concrete type (the name of a generic type is allowed too). The name of the destructor has to be **`=destroy`**.

`=destroy(v)` will be automatically invoked for every local stack variable **`v`** that goes out of scope.

If a structured type features a field with destructable type and the user has not provided an explicit implementation, a destructor for the structured type will be automatically generated. Calls to any base class destructors in both user-defined and generated destructors will be inserted.

A destructor is attached to the type it destructs; expressions of this type can then only be used in *destructible contexts* and as parameters:

```
type
  MyObj = object
    x, y: int
    p: pointer

proc `=destroy`(o: var MyObj) =
  if o.p != nil: dealloc o.p

proc open: MyObj =
  result = MyObj(x: 1, y: 2, p: alloc(3))

proc work(o: MyObj) =
  echo o.x
  # No destructor invoked here for 'o' as 'o' is a parameter.

proc main() =
  # destructor automatically invoked at the end of the scope:
  var x = open()
  # valid: pass 'x' to some other proc:
  work(x)

  # Error: usage of a type with a destructor in a non destructible context
  echo open()
```

A destructible context is currently only the following:

1. The **`expr`** in **`var x = expr`**.
2. The **`expr`** in **`let x = expr`**.
3. The **`expr`** in **`return expr`**.
4. The **`expr`** in **`result = expr`** where **`result`** is the special symbol introduced by the compiler.

These rules ensure that the construction is tied to a variable and can easily be destructed at its scope exit. Later versions of the language will improve the support of destructors.

Be aware that destructors are not called for objects allocated with **`new`**. This may change in future versions of language, but for now the finalizer parameter to **`new`** has to be used.

Note: Destructors are still experimental and the spec might change significantly in order to incorporate an escape analysis.

deepCopy

`=deepCopy` is a builtin that is invoked whenever data is passed to a **`spawn`**'ed proc to ensure memory safety. The programmer can override its behaviour for a specific **`ref`** or **`ptr`** type **`T`**. (Later versions of the language may weaken this restriction.)

The signature has to be:

```
proc `=deepCopy`(x: T): T
```

This mechanism will be used by most data structures that support shared memory like channels to implement thread safe automatic memory management.

The builtin **`deepCopy`** can even clone closures and their environments. See the documentation of spawn for details.

Term rewriting macros

Term rewriting macros are macros or templates that have not only a *name* but also a *pattern* that is searched for after the semantic checking phase of the compiler: This means they provide an easy way to enhance the compilation pipeline with user defined optimizations:

```
template optMul{`*`(a, 2)}(a: int): int = a+a

let x = 3
echo x * 2
```

The compiler now rewrites `x * 2` as `x + x`. The code inside the curlies is the pattern to match against. The operators `*`, `**`, `|`, `~` have a special meaning in patterns if they are written in infix notation, so to match verbatim against `*` the ordinary function call syntax needs to be used.

Unfortunately optimizations are hard to get right and even the tiny example is **wrong**:

```
template optMul{`*`(a, 2)}(a: int): int = a+a

proc f(): int =
  echo "side effect!"
  result = 55

echo f() * 2
```

We cannot duplicate 'a' if it denotes an expression that has a side effect! Fortunately Nim supports side effect analysis:

```
template optMul{`*`(a, 2)}(a: int{noSideEffect}): int = a+a

proc f(): int =
  echo "side effect!"
  result = 55

echo f() * 2 # not optimized ;-)
```

You can make one overload matching with a constraint and one without, and the one with a constraint will have precedence, and so you can handle both cases differently.

So what about `2 * a`? We should tell the compiler `*` is commutative. We cannot really do that however as the following code only swaps arguments blindly:

```
template mulIsCommutative{`*`(a, b)}(a, b: int): int = b*a
```

What optimizers really need to do is a *canonicalization*:

```
template canonMul{`*`(a, b)}(a: int{lit}, b: int): int = b*a
```

The `int{lit}` parameter pattern matches against an expression of type `int`, but only if it's a literal.

Parameter constraints

The parameter constraint expression can use the operators `|` (or), `&` (and) and `~` (not) and the following predicates:

Predicate	Meaning
`atom`	The matching node has no children.

`lit`	The matching node is a literal like "abc", 12.
`sym`	The matching node must be a symbol (a bound identifier).
`ident`	The matching node must be an identifier (an unbound identifier).
`call`	The matching AST must be a call/apply expression.
`lvalue`	The matching AST must be an lvalue.
`sideeffect`	The matching AST must have a side effect.
`nosideeffect`	The matching AST must have no side effect.
`param`	A symbol which is a parameter.
`genericparam`	A symbol which is a generic parameter.
`module`	A symbol which is a module.
`type`	A symbol which is a type.
`var`	A symbol which is a variable.
`let`	A symbol which is a `let` variable.
`const`	A symbol which is a constant.
`result`	The special `result` variable.
`proc`	A symbol which is a proc.
`method`	A symbol which is a method.
`iterator`	A symbol which is an iterator.
`converter`	A symbol which is a converter.
`macro`	A symbol which is a macro.
`template`	A symbol which is a template.
`field`	A symbol which is a field in a tuple or an object.
`enumfield`	A symbol which is a field in an enumeration.
`forvar`	A for loop variable.
`label`	A label (used in `block` statements).
`nk*`	The matching AST must have the specified kind. (Example: `nkIfStmt` denotes an `if` statement.)
`alias`	States that the marked parameter needs to alias with *some* other parameter.
`noalias`	States that *every* other parameter must not alias with the marked parameter.

Predicates that share their name with a keyword have to be escaped with backticks: `` *const* **. The ``alias** and **noalias** predicates refer not only to the matching AST, but also to every other bound parameter; syntactically they need to occur after the ordinary AST predicates:

```
template ex{a = b + c}(a: int{noalias}, b, c: int) =
  # this transformation is only valid if 'b' and 'c' do not alias 'a':
  a = b
  inc a, c
```

Pattern operators

The operators `*`, `**`, `|`, `~` have a special meaning in patterns if they are written in infix notation.

The | operator

The `|` operator if used as infix operator creates an ordered choice:

```
template t{0|1}(): untyped = 3
let a = 1
# outputs 3:
echo a
```

The matching is performed after the compiler performed some optimizations like constant folding, so the following does not work:

```
template t{0|1}(): untyped = 3
# outputs 1:
echo 1
```

The reason is that the compiler already transformed the 1 into "1" for the **echo** statement. However, a term rewriting macro should not change the semantics anyway. In fact they can be deactivated with the **--patterns:off** command line option or temporarily with the **patterns** pragma.

The {} operator

A pattern expression can be bound to a pattern parameter via the **expr{param}** notation:

```
template t{(0|1|2){x}}(x: untyped): untyped = x+1
let a = 1
# outputs 2:
echo a
```

The ~ operator

The **~** operator is the **not** operator in patterns:

```
template t{x = (~x){y} and (~x){z}}(x, y, z: bool) =
  x = y
  if x: x = z

var
  a = false
  b = true
  c = false
a = b and c
echo a
```

The * operator

The * operator can *flatten* a nested binary expression like **a & b & c** to **&(a, b, c)**:

```
var
  calls = 0

proc `&&`(s: varargs[string]): string =
  result = s[0]
  for i in 1..len(s)-1: result.add s[i]
  inc calls

template optConc{ `&&` * a }(a: string): untyped = &&a

let space = " "
echo "my" && (space & "awe" && "some " ) && "concat"

# check that it's been optimized properly:
doAssert calls == 1
```

The second operator of * must be a parameter; it is used to gather all the arguments. The expression **"my" && (space & "awe" && "some ") && "concat"** is passed to **optConc** in **a** as a special list (of kind **nkArgList**) which is flattened into a call expression; thus the invocation of **optConc** produces:

```
`&&`("my", space & "awe", "some ", "concat")
```

The `**` operator

The `**` is much like the `*` operator, except that it gathers not only all the arguments, but also the matched operators in reverse polish notation:

```
import macros

type
  Matrix = object
    dummy: int

proc `*`(a, b: Matrix): Matrix = discard
proc `+`(a, b: Matrix): Matrix = discard
proc `-`(a, b: Matrix): Matrix = discard
proc `$`(a: Matrix): string = result = $a.dummy
proc mat21(): Matrix =
  result.dummy = 21

macro optM{ (`+`|`-`|`*`) ** a }(a: Matrix): untyped =
  echo treeRepr(a)
  result = newCall(bindSym"mat21")

var x, y, z: Matrix

echo x + y * z - x
```

This passes the expression `x + y * z - x` to the `optM` macro as an `nnkArgList` node containing:

```
Arglist
  Sym "x"
  Sym "y"
  Sym "z"
  Sym "*"
  Sym "+"
  Sym "x"
  Sym "-"
```

(Which is the reverse polish notation of `x + y * z - x`.)

Parameters

Parameters in a pattern are type checked in the matching process. If a parameter is of the type `varargs` it is treated specially and it can match 0 or more arguments in the AST to be matched against:

```
template optWrite{
  write(f, x)
  ((write|writeLine){w})(f, y)
}(x, y: varargs[untyped], f: File, w: untyped) =
  w(f, x, y)
```

Example: Partial evaluation

The following example shows how some simple partial evaluation can be implemented with term rewriting:

```
proc p(x, y: int; cond: bool): int =
  result = if cond: x + y else: x - y

template optP1{p(x, y, true)}(x, y: untyped): untyped = x + y
template optP2{p(x, y, false)}(x, y: untyped): untyped = x - y
```

Example: Hoisting

The following example shows how some form of hoisting can be implemented:

```
import pegs

template optPeg{peg(pattern)}(pattern: string{lit}): Peg =
  var gl {.global, gensym.} = peg(pattern)
  gl

for i in 0 .. 3:
  echo match("(a b c)", peg"'(' @ ')'")
  echo match("W_HI_Le", peg"\y 'while'")
```

The **`optPeg`** template optimizes the case of a peg constructor with a string literal, so that the pattern will only be parsed once at program startup and stored in a global **`gl`** which is then re-used. This optimization is called hoisting because it is comparable to classical loop hoisting.

AST based overloading

Parameter constraints can also be used for ordinary routine parameters; these constraints affect ordinary overloading resolution then:

```
proc optLit(a: string{lit|`const`}) =
  echo "string literal"
proc optLit(a: string) =
  echo "no string literal"

const
  constant = "abc"

var
  variable = "xyz"

optLit("literal")
optLit(constant)
optLit(variable)
```

However, the constraints **alias** and **noalias** are not available in ordinary routines.

Move optimization

The **call** constraint is particularly useful to implement a move optimization for types that have copying semantics:

```
proc `[]=`*(t: var Table, key: string, val: string) =
  ## puts a (key, value)-pair into `t`. The semantics of string require
  ## a copy here:
  let idx = findInsertionPosition(key)
  t[idx].key = key
  t[idx].val = val

proc `[]=`*(t: var Table, key: string{call}, val: string{call}) =
  ## puts a (key, value)-pair into `t`. Optimized version that knows that
  ## the strings are unique and thus don't need to be copied:
  let idx = findInsertionPosition(key)
  shallowCopy t[idx].key, key
  shallowCopy t[idx].val, val

var t: Table
# overloading resolution ensures that the optimized []= is called here:
t[f()] = g()
```

Modules

Nim supports splitting a program into pieces by a module concept. Each module needs to be in its own file and has its own namespace. Modules enable information hiding and separate compilation. A module may gain access to symbols of another module by the import statement. Recursive module dependencies are allowed, but slightly subtle. Only top-level symbols that are marked with an asterisk (`*`) are exported. A valid module name can only be a valid Nim identifier (and thus its filename is `identifier.nim`).

The algorithm for compiling modules is:

- compile the whole module as usual, following import statements recursively
- if there is a cycle only import the already parsed symbols (that are exported); if an unknown identifier occurs then abort

This is best illustrated by an example:

```
# Module A
type
  T1* = int  # Module A exports the type ``T1``
import B     # the compiler starts parsing B

proc main() =
  var i = p(3) # works because B has been parsed completely here

main()
```

```
# Module B
import A  # A is not parsed here! Only the already known symbols
          # of A are imported.

proc p*(x: A.T1): A.T1 =
  # this works because the compiler has already
  # added T1 to A's interface symbol table
  result = x + 1
```

Import statement

After the **`import`** statement a list of module names can follow or a single module name followed by an **`except`** list to prevent some symbols to be imported:

```
import strutils except `%`, toUpper

# doesn't work then:
echo "$1" % "abc".toUpper
```

It is not checked that the **`except`** list is really exported from the module. This feature allows to compile against an older version of the module that does not export these identifiers.

Include statement

The **`include`** statement does something fundamentally different than importing a module: it merely includes the contents of a file. The **`include`** statement is useful to split up a large module into several files:

```
include fileA, fileB, fileC
```

Module names in imports

A module alias can be introduced via the **as** keyword:

```
import strutils as su, sequtils as qu

echo su.format("$1", "lalelu")
```

The original module name is then not accessible. The notations **path/to/module** or **path.to.module** or **"path/to/module"** can be used to refer to a module in subdirectories:

```
import lib.pure.strutils, lib/pure/os, "lib/pure/times"
```

Note that the module name is still **strutils** and not **lib.pure.strutils** and so one **cannot** do:

```
import lib.pure.strutils
echo lib.pure.strutils
```

Likewise the following does not make sense as the name is **strutils** already:

```
import lib.pure.strutils as strutils
```

From import statement

After the **from** statement a module name follows followed by an **import** to list the symbols one likes to use without explict full qualification:

```
from strutils import `%`

echo "$1" % "abc"
# always possible: full qualification:
echo strutils.replace("abc", "a", "z")
```

It's also possible to use **from module import nil** if one wants to import the module but wants to enforce fully qualified access to every symbol in **module**.

Export statement

An **export** statement can be used for symbol fowarding so that client modules don't need to import a module's dependencies:

```
# module B
type MyObject* = object
```

```
# module A
import B
export B.MyObject

proc `$`*(x: MyObject): string = "my object"
```

```
# module C
import A

# B.MyObject has been imported implicitly here:
var x: MyObject
echo $x
```

Note on paths

In module related statements, if any part of the module name / path begins with a number, you may have to quote it in double quotes. In the following example, it would be seen as a literal number '3.0' of type 'float64' if not quoted, if uncertain - quote it:

```
import "gfx/3d/somemodule"
```

Scope rules

Identifiers are valid from the point of their declaration until the end of the block in which the declaration occurred. The range where the identifier is known is the scope of the identifier. The exact scope of an identifier depends on the way it was declared.

Block scope

The *scope* of a variable declared in the declaration part of a block is valid from the point of declaration until the end of the block. If a block contains a second block, in which the identifier is redeclared, then inside this block, the second declaration will be valid. Upon leaving the inner block, the first declaration is valid again. An identifier cannot be redefined in the same block, except if valid for procedure or iterator overloading purposes.

Tuple or object scope

The field identifiers inside a tuple or object definition are valid in the following places:

- To the end of the tuple/object definition.
- Field designators of a variable of the given tuple/object type.
- In all descendant types of the object type.

Module scope

All identifiers of a module are valid from the point of declaration until the end of the module. Identifiers from indirectly dependent modules are *not* available. The system module is automatically imported in every module.

If a module imports an identifier by two different modules, each occurrence of the identifier has to be qualified, unless it is an overloaded procedure or iterator in which case the overloading resolution takes place:

```
# Module A
var x*: string
```

```
# Module B
var x*: int
```

```
# Module C
import A, B
write(stdout, x) # error: x is ambiguous
write(stdout, A.x) # no error: qualifier used

var x = 4
write(stdout, x) # not ambiguous: uses the module C's x
```

Compiler Messages

The Nim compiler emits different kinds of messages: hint, warning, and error messages. An *error* message is emitted if the compiler encounters any static error.

Pragmas

Pragmas are Nim's method to give the compiler additional information / commands without introducing a massive number of new keywords. Pragmas are processed on the fly during semantic checking. Pragmas are enclosed in the special `{.` and `.}` curly brackets. Pragmas are also often used as a first implementation to play with a language feature before a nicer syntax to access the feature becomes available.

deprecated pragma

The deprecated pragma is used to mark a symbol as deprecated:

```
proc p() {.deprecated.}
var x {.deprecated.}: char
```

It can also be used as a statement, in that case it takes a list of *renamings*.

```
type
  File = object
  Stream = ref object
{.deprecated: [TFile: File, PStream: Stream].}
```

noSideEffect pragma

The **`noSideEffect`** pragma is used to mark a proc/iterator to have no side effects. This means that the proc/iterator only changes locations that are reachable from its parameters and the return value only depends on the arguments. If none of its parameters have the type `var T` or `ref T` or `ptr T` this means no locations are modified. It is a static error to mark a proc/iterator to have no side effect if the compiler cannot verify this.

As a special semantic rule, the built-in debugEcho (system.html#debugEcho) pretends to be free of side effects, so that it can be used for debugging routines marked as `noSideEffect`.

Future directions: `func` may become a keyword and syntactic sugar for a proc with no side effects:

```
func `+` (x, y: int): int
```

destructor pragma

The **`destructor`** pragma is used to mark a proc to act as a type destructor. Its usage is deprecated, see type bound operations instead.

override pragma

See type bound operations instead.

procvar pragma

The **`procvar`** pragma is used to mark a proc that it can be passed to a procedural variable.

compileTime pragma

The **compileTime** pragma is used to mark a proc or variable to be used at compile time only. No code will be generated for it. Compile time procs are useful as helpers for macros. Since version 0.12.0 of the language, a proc that uses **system.NimNode** within its parameter types is implictly declared **compileTime**:

```
proc astHelper(n: NimNode): NimNode =
  result = n
```

Is the same as:

```
proc astHelper(n: NimNode): NimNode {.compileTime.} =
  result = n
```

noReturn pragma

The **noreturn** pragma is used to mark a proc that never returns.

acyclic pragma

The **acyclic** pragma can be used for object types to mark them as acyclic even though they seem to be cyclic. This is an **optimization** for the garbage collector to not consider objects of this type as part of a cycle:

```
type
  Node = ref NodeObj
  NodeObj {.acyclic, final.} = object
    left, right: Node
    data: string
```

In the example a tree structure is declared with the **Node** type. Note that the type definition is recursive and the GC has to assume that objects of this type may form a cyclic graph. The **acyclic** pragma passes the information that this cannot happen to the GC. If the programmer uses the **acyclic** pragma for data types that are in reality cyclic, the GC may leak memory, but nothing worse happens.

Future directions: The **acyclic** pragma may become a property of a **ref** type:

```
type
  Node = acyclic ref NodeObj
  NodeObj = object
    left, right: Node
    data: string
```

final pragma

The **final** pragma can be used for an object type to specify that it cannot be inherited from.

shallow pragma

The **shallow** pragma affects the semantics of a type: The compiler is allowed to make a shallow copy. This can cause serious semantic issues and break memory safety! However, it can speed up assignments considerably, because the semantics of Nim require deep copying of sequences and strings. This can be expensive, especially if sequences are used to build a tree structure:

```
type
  NodeKind = enum nkLeaf, nkInner
  Node {.final, shallow.} = object
    case kind: NodeKind
    of nkLeaf:
      strVal: string
    of nkInner:
      children: seq[Node]
```

pure pragma

An object type can be marked with the **`pure`** pragma so that its type field which is used for runtime type identification is omitted. This used to be necessary for binary compatibility with other compiled languages.

An enum type can be marked as **`pure`**. Then access of its fields always requires full qualification.

asmNoStackFrame pragma

A proc can be marked with the **`asmNoStackFrame`** pragma to tell the compiler it should not generate a stack frame for the proc. There are also no exit statements like **`return result;`** generated and the generated C function is declared as **`__declspec(naked)`** or **`__attribute__((naked))`** (depending on the used C compiler).

Note: This pragma should only be used by procs which consist solely of assembler statements.

error pragma

The **`error`** pragma is used to make the compiler output an error message with the given content. Compilation does not necessarily abort after an error though.

The **`error`** pragma can also be used to annotate a symbol (like an iterator or proc). The *usage* of the symbol then triggers a compile-time error. This is especially useful to rule out that some operation is valid due to overloading and type conversions:

```
## check that underlying int values are compared and not the pointers:
proc `==`(x, y: ptr int): bool {.error.}
```

fatal pragma

The **`fatal`** pragma is used to make the compiler output an error message with the given content. In contrast to the **`error`** pragma, compilation is guaranteed to be aborted by this pragma. Example:

```
when not defined(objc):
  {.fatal: "Compile this program with the objc command!".}
```

warning pragma

The **`warning`** pragma is used to make the compiler output a warning message with the given content. Compilation continues after the warning.

hint pragma

The **hint** pragma is used to make the compiler output a hint message with the given content. Compilation continues after the hint.

line pragma

The **line** pragma can be used to affect line information of the annotated statement as seen in stack backtraces:

```
template myassert*(cond: untyped, msg = "") =
  if not cond:
    # change run-time line information of the 'raise' statement:
    {.line: InstantiationInfo().}:
      raise newException(EAssertionFailed, msg)
```

If the **line** pragma is used with a parameter, the parameter needs be a **tuple[filename: string, line: int]**. If it is used without a parameter, **system.InstantiationInfo()** is used.

linearScanEnd pragma

The **linearScanEnd** pragma can be used to tell the compiler how to compile a Nim case statement. Syntactically it has to be used as a statement:

```
case myInt
of 0:
  echo "most common case"
of 1:
  {.linearScanEnd.}
  echo "second most common case"
of 2: echo "unlikely: use branch table"
else: echo "unlikely too: use branch table for ", myInt
```

In the example, the case branches **0** and **1** are much more common than the other cases. Therefore the generated assembler code should test for these values first, so that the CPU's branch predictor has a good chance to succeed (avoiding an expensive CPU pipeline stall). The other cases might be put into a jump table for O(1) overhead, but at the cost of a (very likely) pipeline stall.

The **linearScanEnd** pragma should be put into the last branch that should be tested against via linear scanning. If put into the last branch of the whole **case** statement, the whole **case** statement uses linear scanning.

computedGoto pragma

The **computedGoto** pragma can be used to tell the compiler how to compile a Nim case in a **while true** statement. Syntactically it has to be used as a statement inside the loop:

```
type
  MyEnum = enum
    enumA, enumB, enumC, enumD, enumE

proc vm() =
  var instructions: array [0..100, MyEnum]
  instructions[2] = enumC
  instructions[3] = enumD
  instructions[4] = enumA
  instructions[5] = enumD
  instructions[6] = enumC
  instructions[7] = enumA
  instructions[8] = enumB

  instructions[12] = enumE
  var pc = 0
  while true:
    {.computedGoto.}
    let instr = instructions[pc]
    case instr
    of enumA:
      echo "yeah A"
    of enumC, enumD:
      echo "yeah CD"
    of enumB:
      echo "yeah B"
    of enumE:
      break
    inc(pc)

vm()
```

As the example shows **computedGoto** is mostly useful for interpreters. If the underlying backend (C compiler) does not support the computed goto extension the pragma is simply ignored.

unroll pragma

The **unroll** pragma can be used to tell the compiler that it should unroll a for or while loop for runtime efficiency:

```
proc searchChar(s: string, c: char): int =
  for i in 0 .. s.high:
    {.unroll: 4.}
    if s[i] == c: return i
  result = -1
```

In the above example, the search loop is unrolled by a factor 4. The unroll factor can be left out too; the compiler then chooses an appropriate unroll factor.

Note: Currently the compiler recognizes but ignores this pragma.

immediate pragma

See Ordinary vs immediate templates.

compilation option pragmas

The listed pragmas here can be used to override the code generation options for a proc/method /converter.

The implementation currently provides the following possible options (various others may be added later).

pragma	allowed values	description
checks	on|off	Turns the code generation for all runtime checks on or off.
boundChecks	on|off	Turns the code generation for array bound checks on or off.
overflowChecks	on|off	Turns the code generation for over- or underflow checks on or off.
nilChecks	on|off	Turns the code generation for nil pointer checks on or off.
assertions	on|off	Turns the code generation for assertions on or off.
warnings	on|off	Turns the warning messages of the compiler on or off.
hints	on|off	Turns the hint messages of the compiler on or off.
optimization	none|speed|size	Optimize the code for speed or size, or disable optimization.
patterns	on|off	Turns the term rewriting templates/macros on or off.
callconv	cdecl|...	Specifies the default calling convention for all procedures (and procedure types) that follow.

Example:

```
{.checks: off, optimization: speed.}
# compile without runtime checks and optimize for speed
```

push and pop pragmas

The push/pop pragmas are very similar to the option directive, but are used to override the settings temporarily. Example:

```
{.push checks: off.}
# compile this section without runtime checks as it is
# speed critical
# ... some code ...
{.pop.} # restore old settings
```

register pragma

The `register` pragma is for variables only. It declares the variable as `register`, giving the compiler a hint that the variable should be placed in a hardware register for faster access. C compilers usually ignore this though and for good reasons: Often they do a better job without it anyway.

In highly specific cases (a dispatch loop of a bytecode interpreter for example) it may provide benefits, though.

global pragma

The `global` pragma can be applied to a variable within a proc to instruct the compiler to store it in a global location and initialize it once at program startup.

```
proc isHexNumber(s: string): bool =
  var pattern {.global.} = re"[0-9a-fA-F]+"
  result = s.match(pattern)
```

When used within a generic proc, a separate unique global variable will be created for each instantiation of the proc. The order of initialization of the created global variables within a module is not defined, but all of them will be initialized after any top-level variables in their originating module and before any variable in a module that imports it.

deadCodeElim pragma

The **`deadCodeElim`** pragma only applies to whole modules: It tells the compiler to activate (or deactivate) dead code elimination for the module the pragma appears in.

The **`--deadCodeElim:on`** command line switch has the same effect as marking every module with **`{.deadCodeElim:on}`**. However, for some modules such as the GTK wrapper it makes sense to *always* turn on dead code elimination - no matter if it is globally active or not.

Example:

```
{.deadCodeElim: on.}
```

pragma pragma

The **`pragma`** pragma can be used to declare user defined pragmas. This is useful because Nim's templates and macros do not affect pragmas. User defined pragmas are in a different module-wide scope than all other symbols. They cannot be imported from a module.

Example:

```
when appType == "lib":
  {.pragma: rtl, exportc, dynlib, cdecl.}
else:
  {.pragma: rtl, importc, dynlib: "client.dll", cdecl.}

proc p*(a, b: int): int {.rtl.} =
  result = a+b
```

In the example a new pragma named **`rtl`** is introduced that either imports a symbol from a dynamic library or exports the symbol for dynamic library generation.

Disabling certain messages

Nim generates some warnings and hints ("line too long") that may annoy the user. A mechanism for disabling certain messages is provided: Each hint and warning message contains a symbol in brackets. This is the message's identifier that can be used to enable or disable it:

```
{.hint[LineTooLong]: off.} # turn off the hint about too long lines
```

This is often better than disabling all warnings at once.

used pragma

Nim produces a warning for symbols that are not exported and not used either. The **`used`** pragma can be attached to a symbol to suppress this warning. This is particularly useful when the symbol was generated by a macro:

```
template implementArithOps(T) =
  proc echoAdd(a, b: T) {.used.} =
    echo a + b
  proc echoSub(a, b: T) {.used.} =
    echo a - b

# no warning produced for the unused 'echoSub'
implementArithOps(int)
echoAdd 3, 5
```

experimental pragma

The **experimental** pragma enables experimental language features. Depending on the concrete feature this means that the feature is either considered too unstable for an otherwise stable release or that the future of the feature is uncertain (it may be removed any time).

Example:

```
{.experimental.}
type
  FooId = distinct int
  BarId = distinct int
using
  foo: FooId
  bar: BarId

proc useUsing(bar, foo) =
  echo "bar is of type BarId"
  echo "foo is of type FooId"
```

Implementation Specific Pragmas

This section describes additional pragmas that the current Nim implementation supports but which should not be seen as part of the language specification.

Bitsize pragma

The **bitsize** pragma is for object field members. It declares the field as a bitfield in C/C++.

```
type
  mybitfield = object
    flag {.bitsize:1.}: cuint
```

generates:

```
struct mybitfield {
  unsigned int flag:1;
};
```

Volatile pragma

The **volatile** pragma is for variables only. It declares the variable as **volatile**, whatever that means in C/C++ (its semantics are not well defined in C/C++).

Note: This pragma will not exist for the LLVM backend.

NoDecl pragma

The **noDecl** pragma can be applied to almost any symbol (variable, proc, type, etc.) and is sometimes useful for interoperability with C: It tells Nim that it should not generate a declaration for the symbol in the C code. For example:

```
var
  EACCES {.importc, noDecl.}: cint # pretend EACCES was a variable, as
                                   # Nim does not know its value
```

However, the **header** pragma is often the better alternative.

Note: This will not work for the LLVM backend.

Header pragma

The **header** pragma is very similar to the **noDecl** pragma: It can be applied to almost any symbol and specifies that it should not be declared and instead the generated code should contain an **#include**:

```
type
  PFile {.importc: "FILE*", header: "<stdio.h>".} = distinct pointer
    # import C's FILE* type; Nim will treat it as a new pointer type
```

The **header** pragma always expects a string constant. The string contant contains the header file: As usual for C, a system header file is enclosed in angle brackets: **<>**. If no angle brackets are given, Nim encloses the header file in **""** in the generated C code.

Note: This will not work for the LLVM backend.

IncompleteStruct pragma

The **incompleteStruct** pragma tells the compiler to not use the underlying C **struct** in a **sizeof** expression:

```
type
  DIR* {.importc: "DIR", header: "<dirent.h>",
         final, pure, incompleteStruct.} = object
```

Compile pragma

The **compile** pragma can be used to compile and link a C/C++ source file with the project:

```
{.compile: "myfile.cpp".}
```

Note: Nim computes a SHA1 checksum and only recompiles the file if it has changed. You can use the **-f** command line option to force recompilation of the file.

Link pragma

The **link** pragma can be used to link an additional file with the project:

```
{.link: "myfile.o".}
```

PassC pragma

The **passC** pragma can be used to pass additional parameters to the C compiler like you would using the commandline switch **--passC**:

```
{.passC: "-Wall -Werror".}
```

Note that you can use **gorge** from the system module (system.html) to embed parameters from an external command at compile time:

```
{.passC: gorge("pkg-config --cflags sdl").}
```

PassL pragma

The **passL** pragma can be used to pass additional parameters to the linker like you would using the commandline switch **--passL**:

```
{.passL: "-lSDLmain -lSDL".}
```

Note that you can use **gorge** from the system module (system.html) to embed parameters from an external command at compile time:

```
{.passL: gorge("pkg-config --libs sdl").}
```

Emit pragma

The **emit** pragma can be used to directly affect the output of the compiler's code generator. So it makes your code unportable to other code generators/backends. Its usage is highly discouraged! However, it can be extremely useful for interfacing with C++ or Objective C code.

Example:

```
{.emit: """
static int cvariable = 420;
""".}

{.push stackTrace:off.}
proc embedsC() =
  var nimVar = 89
  # access Nim symbols within an emit section outside of string literals:
  {.emit: ["""fprintf(stdout, "%d\n", cvariable + (int)""", nimVar, ");"].}
{.pop.}

embedsC()
```

For backwards compatibility, if the argument to the **`emit`** statement is a single string literal, Nim symbols can be referred to via backticks. This usage is however deprecated.

For a toplevel emit statement the section where in the generated C/C++ file the code should be emitted can be influenced via the prefixes **`/*TYPESECTION*/`** or **`/*VARSECTION*/`** or **`/*INCLUDESECTION*/`**:

```
{.emit: """/*TYPESECTION*/
struct Vector3 {
public:
  Vector3(): x(5) {}
  Vector3(float x_): x(x_) {}
  float x;
};
""".}

type Vector3 {.importcpp: "Vector3", nodecl} = object
  x: cfloat

proc constructVector3(a: cfloat): Vector3 {.importcpp: "Vector3(@)", nodecl}
```

ImportCpp pragma

Note: c2nim (c2nim.html) can parse a large subset of C++ and knows about the **`importcpp`** pragma pattern language. It is not necessary to know all the details described here.

Similar to the importc pragma for C (manual.html#importc-pragma), the **`importcpp`** pragma can be used to import C++ methods or C++ symbols in general. The generated code then uses the C++ method calling syntax: **`obj->method(arg)`**. In combination with the **`header`** and **`emit`** pragmas this allows *sloppy* interfacing with libraries written in C++:

```
# Horrible example of how to interface with a C++ engine ... ;-)

{.link: "/usr/lib/libIrrlicht.so".}

{.emit: """
using namespace irr;
using namespace core;
using namespace scene;
using namespace video;
using namespace io;
using namespace gui;
""".}

const
  irr = "<irrlicht/irrlicht.h>"

type
  IrrlichtDeviceObj {.final, header: irr,
                      importcpp: "IrrlichtDevice".} = object
  IrrlichtDevice = ptr IrrlichtDeviceObj

proc createDevice(): IrrlichtDevice {.
  header: irr, importcpp: "createDevice(@)".}
proc run(device: IrrlichtDevice): bool {.
  header: irr, importcpp: "#.run(@)".}
```

The compiler needs to be told to generate C++ (command **cpp**) for this to work. The conditional symbol **cpp** is defined when the compiler emits C++ code.

Namespaces

The *sloppy interfacing* example uses **.emit** to produce **using namespace** declarations. It is usually much better to instead refer to the imported name via the **namespace::identifier** notation:

```
type
  IrrlichtDeviceObj {.final, header: irr,
                      importcpp: "irr::IrrlichtDevice".} = object
```

Importcpp for enums

When **importcpp** is applied to an enum type the numerical enum values are annotated with the C++ enum type, like in this example: **((TheCppEnum)(3))**. (This turned out to be the simplest way to implement it.)

Importcpp for procs

Note that the **importcpp** variant for procs uses a somewhat cryptic pattern language for maximum flexibility:

- A hash **#** symbol is replaced by the first or next argument.
- A dot following the hash **#.** indicates that the call should use C++'s dot or arrow notation.
- An at symbol **@** is replaced by the remaining arguments, separated by commas.

For example:

```
proc cppMethod(this: CppObj, a, b, c: cint) {.importcpp: "#.CppMethod(@)".}
var x: ptr CppObj
cppMethod(x[], 1, 2, 3)
```

Produces:

```
x->CppMethod(1, 2, 3)
```

As a special rule to keep backwards compatibility with older versions of the **importcpp** pragma, if there is no special pattern character (any of `# ' @`) at all, C++'s dot or arrow notation is assumed, so the above example can also be written as:

```
proc cppMethod(this: CppObj, a, b, c: cint) {.importcpp: "CppMethod".}
```

Note that the pattern language naturally also covers C++'s operator overloading capabilities:

```
proc vectorAddition(a, b: Vec3): Vec3 {.importcpp: "# + #".}
proc dictLookup(a: Dict, k: Key): Value {.importcpp: "#[#]".}
```

- An apostrophe `'` followed by an integer `i` in the range 0..9 is replaced by the i'th parameter *type*. The 0th position is the result type. This can be used to pass types to C++ function templates. Between the `'` and the digit an asterisk can be used to get to the base type of the type. (So it "takes away a star" from the type; `T*` becomes `T`.) Two stars can be used to get to the element type of the element type etc.

For example:

```
type Input {.importcpp: "System::Input".} = object
proc getSubsystem*[T](): ptr T {.importcpp: "SystemManager::getSubsystem<'*0>()", nodecl.}

let x: ptr Input = getSubsystem[Input]()
```

Produces:

```
x = SystemManager::getSubsystem<System::Input>()
```

- `#@` is a special case to support a **cnew** operation. It is required so that the call expression is inlined directly, without going through a temporary location. This is only required to circumvent a limitation of the current code generator.

For example C++'s **new** operator can be "imported" like this:

```
proc cnew*[T](x: T): ptr T {.importcpp: "(new '*0#@)", nodecl.}

# constructor of 'Foo':
proc constructFoo(a, b: cint): Foo {.importcpp: "Foo(@)".}

let x = cnew constructFoo(3, 4)
```

Produces:

```
x = new Foo(3, 4)
```

However, depending on the use case `new Foo` can also be wrapped like this instead:

```
proc newFoo(a, b: cint): ptr Foo {.importcpp: "new Foo(@)".}

let x = newFoo(3, 4)
```

Wrapping constructors

Sometimes a C++ class has a private copy constructor and so code like **`Class c = Class(1,2);`** must not be generated but instead **`Class c(1,2);`**. For this purpose the Nim proc that wraps a C++ constructor needs to be annotated with the constructor pragma. This pragma also helps to generate faster C++ code since construction then doesn't invoke the copy constructor:

```
# a better constructor of 'Foo':
proc constructFoo(a, b: cint): Foo {.importcpp: "Foo(@)", constructor.}
```

Wrapping destructors

Since Nim generates C++ directly, any destructor is called implicitly by the C++ compiler at the scope exits. This means that often one can get away with not wrapping the destructor at all! However when it needs to be invoked explicitly, it needs to be wrapped. But the pattern language already provides everything that is required for that:

```
proc destroyFoo(this: var Foo) {.importcpp: "#.~Foo()".}
```

Importcpp for objects

Generic **`importcpp`**'ed objects are mapped to C++ templates. This means that you can import C++'s templates rather easily without the need for a pattern language for object types:

```
type
  StdMap {.importcpp: "std::map", header: "<map>".} [K, V] = object
proc `[]=`[K, V](this: var StdMap[K, V]; key: K; val: V) {.
  importcpp: "#[#] = #", header: "<map>".}

var x: StdMap[cint, cdouble]
x[6] = 91.4
```

Produces:

```
std::map<int, double> x;
x[6] = 91.4;
```

- If more precise control is needed, the apostrophe **`'`** can be used in the supplied pattern to denote the concrete type parameters of the generic type. See the usage of the apostrophe operator in proc patterns for more details.

```
type
  VectorIterator {.importcpp: "std::vector<'0>::iterator".} [T] = object

var x: VectorIterator[cint]
```

Produces:

```
std::vector<int>::iterator x;
```

ImportObjC pragma

Similar to the importc pragma for C (manual.html#importc-pragma), the **importobjc** pragma can be used to import Objective C methods. The generated code then uses the Objective C method calling syntax: **[obj method param1: arg]**. In addition with the **header** and **emit** pragmas this allows *sloppy* interfacing with libraries written in Objective C:

```
# horrible example of how to interface with GNUStep ...

{.passL: "-lobjc".}
{.emit: """
#include <objc/Object.h>
@interface Greeter:Object
{
}

- (void)greet:(long)x y:(long)dummy;
@end

#include <stdio.h>
@implementation Greeter

- (void)greet:(long)x y:(long)dummy
{
  printf("Hello, World!\n");
}
@end

#include <stdlib.h>
""".}

type
  Id {.importc: "id", header: "<objc/Object.h>", final.} = distinct int

proc newGreeter: Id {.importobjc: "Greeter new", nodecl.}
proc greet(self: Id, x, y: int) {.importobjc: "greet", nodecl.}
proc free(self: Id) {.importobjc: "free", nodecl.}

var g = newGreeter()
g.greet(12, 34)
g.free()
```

The compiler needs to be told to generate Objective C (command **objc**) for this to work. The conditional symbol **objc** is defined when the compiler emits Objective C code.

CodegenDecl pragma

The **codegenDecl** pragma can be used to directly influence Nim's code generator. It receives a format string that determines how the variable or proc is declared in the generated code.

For variables $1 in the format string represents the type of the variable and $2 is the name of the variable.

The following Nim code:

```
var
  a {.codegenDecl: "$# progmem $#".}: int
```

will generate this C code:

```
int progmem a
```

For procedures $1 is the return type of the procedure, $2 is the name of the procedure and $3 is the parameter list.

The following nim code:

```
proc myinterrupt() {.codegenDecl: "__interrupt $# $#$#".} =
  echo "realistic interrupt handler"
```

will generate this code:

```
__interrupt void myinterrupt()
```

InjectStmt pragma

The **injectStmt** pragma can be used to inject a statement before every other statement in the current module. It is only supposed to be used for debugging:

```
{.injectStmt: gcInvariants().}

# ... complex code here that produces crashes ...
```

compile time define pragmas

The pragmas listed here can be used to optionally accept values from the -d/--define option at compile time.

The implementation currently provides the following possible options (various others may be added later).

pragma	description
intdefine	Reads in a build-time define as an integer
strdefine	Reads in a build-time define as a string

```
const FooBar {.intdefine.}: int = 5
echo FooBar
```

```
nim c -d:FooBar=42 foobar.c
```

In the above example, providing the -d flag causes the symbol **FooBar** to be overwritten at compile time, printing out 42. If the **-d:FooBar=42** were to be omitted, the default value of 5 would be used.

Foreign function interface

Nim's FFI (foreign function interface) is extensive and only the parts that scale to other future backends (like the LLVM/JavaScript backends) are documented here.

Importc pragma

The **importc** pragma provides a means to import a proc or a variable from C. The optional argument is a string containing the C identifier. If the argument is missing, the C name is the Nim identifier *exactly as spelled*:

```
proc printf(formatstr: cstring) {.header: "<stdio.h>", importc: "printf", varargs.}
```

Note that this pragma is somewhat of a misnomer: Other backends do provide the same feature under the same name. Also, if one is interfacing with C++ the ImportCpp pragma (manual.html#implementation-specific-pragmas-importcpp-pragma) and interfacing with Objective-C the ImportObjC pragma (manual.html#implementation-specific-pragmas-importobjc-pragma) can be used.

The string literal passed to **importc** can be a format string:

```
proc p(s: cstring) {.importc: "prefix$1".}
```

In the example the external name of **p** is set to **prefixp**. Only **$1** is available and a literal dollar sign must be written as **$$**.

Exportc pragma

The **exportc** pragma provides a means to export a type, a variable, or a procedure to C. Enums and constants can't be exported. The optional argument is a string containing the C identifier. If the argument is missing, the C name is the Nim identifier *exactly as spelled*:

```
proc callme(formatstr: cstring) {.exportc: "callMe", varargs.}
```

Note that this pragma is somewhat of a misnomer: Other backends do provide the same feature under the same name.

The string literal passed to **exportc** can be a format string:

```
proc p(s: string) {.exportc: "prefix$1".} =
  echo s
```

In the example the external name of **p** is set to **prefixp**. Only **$1** is available and a literal dollar sign must be written as **$$**.

Extern pragma

Like **exportc** or **importc**, the **extern** pragma affects name mangling. The string literal passed to **extern** can be a format string:

```
proc p(s: string) {.extern: "prefix$1".} =
  echo s
```

In the example the external name of **p** is set to **prefixp**. Only **$1** is available and a literal dollar sign must be written as **$$**.

Bycopy pragma

The **bycopy** pragma can be applied to an object or tuple type and instructs the compiler to pass the type by value to procs:

```
type
  Vector {.bycopy, pure.} = object
    x, y, z: float
```

Byref pragma

The **byref** pragma can be applied to an object or tuple type and instructs the compiler to pass the type by reference (hidden pointer) to procs.

Varargs pragma

The **varargs** pragma can be applied to procedures only (and procedure types). It tells Nim that the proc can take a variable number of parameters after the last specified parameter. Nim string values will be converted to C strings automatically:

```
proc printf(formatstr: cstring) {.nodecl, varargs.}

printf("hallo %s", "world") # "world" will be passed as C string
```

Union pragma

The **union** pragma can be applied to any **object** type. It means all of the object's fields are overlaid in memory. This produces a **union** instead of a **struct** in the generated C/C++ code. The object declaration then must not use inheritance or any GC'ed memory but this is currently not checked.

Future directions: GC'ed memory should be allowed in unions and the GC should scan unions conservatively.

Packed pragma

The **packed** pragma can be applied to any **object** type. It ensures that the fields of an object are packed back-to-back in memory. It is useful to store packets or messages from/to network or hardware drivers, and for interoperability with C. Combining packed pragma with inheritance is not defined, and it should not be used with GC'ed memory (ref's).

Future directions: Using GC'ed memory in packed pragma will result in compile-time error. Usage with inheritance should be defined and documented.

Unchecked pragma

The **unchecked** pragma can be used to mark a named array as **unchecked** meaning its bounds are not checked. This is often useful to implement customized flexibly sized arrays. Additionally an unchecked array is translated into a C array of undetermined size:

```
type
  ArrayPart{.unchecked.} = array[0, int]
  MySeq = object
    len, cap: int
    data: ArrayPart
```

Produces roughly this C code:

```
typedef struct {
  NI len;
  NI cap;
  NI data[];
} MySeq;
```

The base type of the unchecked array may not contain any GC'ed memory but this is currently not checked.

Future directions: GC'ed memory should be allowed in unchecked arrays and there should be an explicit annotation of how the GC is to determine the runtime size of the array.

Dynlib pragma for import

With the **`dynlib`** pragma a procedure or a variable can be imported from a dynamic library (**`.dll`** files for Windows, **`lib*.so`** files for UNIX). The non-optional argument has to be the name of the dynamic library:

```
proc gtk_image_new(): PGtkWidget
  {.cdecl, dynlib: "libgtk-x11-2.0.so", importc.}
```

In general, importing a dynamic library does not require any special linker options or linking with import libraries. This also implies that no *devel* packages need to be installed.

The **`dynlib`** import mechanism supports a versioning scheme:

```
proc Tcl_Eval(interp: pTcl_Interp, script: cstring): int {.cdecl,
  importc, dynlib: "libtcl(|8.5|8.4|8.3).so.(1|0)".}
```

At runtime the dynamic library is searched for (in this order):

```
libtcl.so.1
libtcl.so.0
libtcl8.5.so.1
libtcl8.5.so.0
libtcl8.4.so.1
libtcl8.4.so.0
libtcl8.3.so.1
libtcl8.3.so.0
```

The **`dynlib`** pragma supports not only constant strings as argument but also string expressions in general:

```
import os

proc getDllName: string =
  result = "mylib.dll"
  if existsFile(result): return
  result = "mylib2.dll"
  if existsFile(result): return
  quit("could not load dynamic library")

proc myImport(s: cstring) {.cdecl, importc, dynlib: getDllName().}
```

Note: Patterns like `libtcl(|8.5|8.4).so` are only supported in constant strings, because they are precompiled.

Note: Passing variables to the `dynlib` pragma will fail at runtime because of order of initialization problems.

Note: A `dynlib` import can be overriden with the `--dynlibOverride:name` command line option. The Compiler User Guide contains further information.

Dynlib pragma for export

With the `dynlib` pragma a procedure can also be exported to a dynamic library. The pragma then has no argument and has to be used in conjunction with the `exportc` pragma:

```
proc exportme(): int {.cdecl, exportc, dynlib.}
```

This is only useful if the program is compiled as a dynamic library via the `--app:lib` command line option. This pragma only has an effect for the code generation on the Windows target, so when this pragma is forgotten and the dynamic library is only tested on Mac and/or Linux, there won't be an error. On Windows this pragma adds `__declspec(dllexport)` to the function declaration.

Threads

To enable thread support the **--threads:on** command line switch needs to be used. The **system** module then contains several threading primitives. See the threads (threads.html) and channels (channels.html) modules for the low level thread API. There are also high level parallelism constructs available. See spawn for further details.

Nim's memory model for threads is quite different than that of other common programming languages (C, Pascal, Java): Each thread has its own (garbage collected) heap and sharing of memory is restricted to global variables. This helps to prevent race conditions. GC efficiency is improved quite a lot, because the GC never has to stop other threads and see what they reference. Memory allocation requires no lock at all! This design easily scales to massive multicore processors that are becoming the norm.

Thread pragma

A proc that is executed as a new thread of execution should be marked by the **thread** pragma for reasons of readability. The compiler checks for violations of the no heap sharing restriction: This restriction implies that it is invalid to construct a data structure that consists of memory allocated from different (thread local) heaps.

A thread proc is passed to **createThread** or **spawn** and invoked indirectly; so the **thread** pragma implies **procvar**.

GC safety

We call a proc **p** GC safe when it doesn't access any global variable that contains GC'ed memory (**string**, **seq**, **ref** or a closure) either directly or indirectly through a call to a GC unsafe proc.

The gcsafe annotation can be used to mark a proc to be gcsafe, otherwise this property is inferred by the compiler. Note that **noSideEffect** implies **gcsafe**. The only way to create a thread is via **spawn** or **createThread**. **spawn** is usually the preferable method. Either way the invoked proc must not use **var** parameters nor must any of its parameters contain a **ref** or **closure** type. This enforces the *no heap sharing restriction*.

Routines that are imported from C are always assumed to be **gcsafe**. To disable the GC-safety checking the **--threadAnalysis:off** command line switch can be used. This is a temporary workaround to ease the porting effort from old code to the new threading model.

To override the compiler's gcsafety analysis a **{.gcsafe.}** pragma block can be used:

```
var
  someGlobal: string = "some string here"
  perThread {.threadvar.}: string

proc setPerThread() =
  {.gcsafe.}:
    deepCopy(perThread, someGlobal)
```

Future directions:

- A shared GC'ed heap might be provided.

Threadvar pragma

A global variable can be marked with the **threadvar** pragma; it is a thread-local variable then:

```
var checkpoints* {.threadvar.}: seq[string]
```

Due to implementation restrictions thread local variables cannot be initialized within the `var` section. (Every thread local variable needs to be replicated at thread creation.)

Threads and exceptions

The interaction between threads and exceptions is simple: A *handled* exception in one thread cannot affect any other thread. However, an *unhandled* exception in one thread terminates the whole *process*!

Parallel & Spawn

Nim has two flavors of parallelism:

1. Structured parallelism via the **`parallel`** statement.
2. Unstructured parallelism via the standalone **`spawn`** statement.

Nim has a builtin thread pool that can be used for CPU intensive tasks. For IO intensive tasks the **`async`** and **`await`** features should be used instead. Both parallel and spawn need the threadpool (threadpool.html) module to work.

Somewhat confusingly, **`spawn`** is also used in the **`parallel`** statement with slightly different semantics. **`spawn`** always takes a call expression of the form **`f(a, ...)`**. Let **`T`** be **`f`**'s return type. If **`T`** is **`void`** then **`spawn`**'s return type is also **`void`** otherwise it is **`FlowVar[T]`**.

Within a **`parallel`** section sometimes the **`FlowVar[T]`** is eliminated to **`T`**. This happens when **`T`** does not contain any GC'ed memory. The compiler can ensure the location in **`location = spawn f(...)`** is not read prematurely within a **`parallel`** section and so there is no need for the overhead of an indirection via **`FlowVar[T]`** to ensure correctness.

Note: Currently exceptions are not propagated between **`spawn`**'ed tasks!

Spawn statement

spawn can be used to pass a task to the thread pool:

```
import threadpool

proc processLine(line: string) =
  discard "do some heavy lifting here"

for x in lines("myinput.txt"):
  spawn processLine(x)
sync()
```

For reasons of type safety and implementation simplicity the expression that **`spawn`** takes is restricted:

- It must be a call expression **`f(a, ...)`**.
- **`f`** must be **`gcsafe`**.
- **`f`** must not have the calling convention **`closure`**.
- **`f`**'s parameters may not be of type **`var`**. This means one has to use raw **`ptr`**'s for data passing reminding the programmer to be careful.
- **`ref`** parameters are deeply copied which is a subtle semantic change and can cause performance problems but ensures memory safety. This deep copy is performed via **`system.deepCopy`** and so can be overridden.
- For *safe* data exchange between **`f`** and the caller a global **`TChannel`** needs to be used. However, since spawn can return a result, often no further communication is required.

`spawn` executes the passed expression on the thread pool and returns a data flow variable **`FlowVar[T]`** that can be read from. The reading with the **`^`** operator is **blocking**. However, one can use **`awaitAny`** to wait on multiple flow variables at the same time:

```
import threadpool, ...

# wait until 2 out of 3 servers received the update:
proc main =
  var responses = newSeq[FlowVarBase](3)
  for i in 0..2:
    responses[i] = spawn tellServer(Update, "key", "value")
  var index = awaitAny(responses)
  assert index >= 0
  responses.del(index)
  discard awaitAny(responses)
```

Data flow variables ensure that no data races are possible. Due to technical limitations not every type **T** is possible in a data flow variable: **T** has to be of the type **ref**, **string**, **seq** or of a type that doesn't contain a type that is garbage collected. This restriction is not hard to work-around in practice.

Parallel statement

Example:

```
# Compute PI in an inefficient way
import strutils, math, threadpool

proc term(k: float): float = 4 * math.pow(-1, k) / (2*k + 1)

proc pi(n: int): float =
  var ch = newSeq[float](n+1)
  parallel:
    for k in 0..ch.high:
      ch[k] = spawn term(float(k))
  for k in 0..ch.high:
    result += ch[k]

echo formatFloat(pi(5000))
```

The parallel statement is the preferred mechanism to introduce parallelism in a Nim program. A subset of the Nim language is valid within a **parallel** section. This subset is checked to be free of data races at compile time. A sophisticated disjoint checker ensures that no data races are possible even though shared memory is extensively supported!

The subset is in fact the full language with the following restrictions / changes:

- **spawn** within a **parallel** section has special semantics.
- Every location of the form **a[i]** and **a[i..j]** and **dest** where **dest** is part of the pattern **dest = spawn f(...)** has to be provably disjoint. This is called the *disjoint check*.
- Every other complex location **loc** that is used in a spawned proc (**spawn f(loc)**) has to be immutable for the duration of the **parallel** section. This is called the *immutability check*. Currently it is not specified what exactly "complex location" means. We need to make this an optimization!
- Every array access has to be provably within bounds. This is called the *bounds check*.
- Slices are optimized so that no copy is performed. This optimization is not yet performed for ordinary slices outside of a **parallel** section.

Guards and locks

Apart from **spawn** and **parallel** Nim also provides all the common low level concurrency mechanisms like locks, atomic intristics or condition variables.

Nim significantly improves on the safety of these features via additional pragmas:

1. A guard annotation is introduced to prevent data races.
2. Every access of a guarded memory location needs to happen in an appropriate locks statement.
3. Locks and routines can be annotated with lock levels to prevent deadlocks at compile time.

Guards and the locks section

Protecting global variables

Object fields and global variables can be annotated via a **guard** pragma:

```
var glock: TLock
var gdata {.guard: glock.}: int
```

The compiler then ensures that every access of **gdata** is within a **locks** section:

```
proc invalid =
  # invalid: unguarded access:
  echo gdata

proc valid =
  # valid access:
  {.locks: [glock].}:
    echo gdata
```

Top level accesses to **gdata** are always allowed so that it can be initialized conveniently. It is *assumed* (but not enforced) that every top level statement is executed before any concurrent action happens.

The **locks** section deliberately looks ugly because it has no runtime semantics and should not be used directly! It should only be used in templates that also implement some form of locking at runtime:

```
template lock(a: TLock; body: untyped) =
  pthread_mutex_lock(a)
  {.locks: [a].}:
    try:
      body
    finally:
      pthread_mutex_unlock(a)
```

The guard does not need to be of any particular type. It is flexible enough to model low level lockfree mechanisms:

```
var dummyLock {.compileTime.}: int
var atomicCounter {.guard: dummyLock.}: int

template atomicRead(x): untyped =
  {.locks: [dummyLock].}:
    memoryReadBarrier()
    x

echo atomicRead(atomicCounter)
```

The **locks** pragma takes a list of lock expressions **locks: [a, b, ...]** in order to support *multi lock* statements. Why these are essential is explained in the lock levels section.

Protecting general locations

The **guard** annotation can also be used to protect fields within an object. The guard then needs to be another field within the same object or a global variable.

Since objects can reside on the heap or on the stack this greatly enhances the expressivity of the language:

```
type
  ProtectedCounter = object
    v {.guard: L.}: int
    L: TLock

proc incCounters(counters: var openArray[ProtectedCounter]) =
  for i in 0..counters.high:
    lock counters[i].L:
      inc counters[i].v
```

The access to field **x.v** is allowed since its guard **x.L** is active. After template expansion, this amounts to:

```
proc incCounters(counters: var openArray[ProtectedCounter]) =
  for i in 0..counters.high:
    pthread_mutex_lock(counters[i].L)
    {.locks: [counters[i].L].}:
      try:
        inc counters[i].v
      finally:
        pthread_mutex_unlock(counters[i].L)
```

There is an analysis that checks that **counters[i].L** is the lock that corresponds to the protected location **counters[i].v**. This analysis is called path analysis because it deals with paths to locations like **obj.field[i].fieldB[j]**.

The path analysis is **currently unsound**, but that doesn't make it useless. Two paths are considered equivalent if they are syntactically the same.

This means the following compiles (for now) even though it really should not:

```
{.locks: [a[i].L].}:
  inc i
  access a[i].v
```

Lock levels

Lock levels are used to enforce a global locking order in order to prevent deadlocks at compile-time. A lock level is an constant integer in the range 0..1_000. Lock level 0 means that no lock is acquired at all.

If a section of code holds a lock of level `M` than it can also acquire any lock of level **`N < M`**. Another lock of level `M` cannot be acquired. Locks of the same level can only be acquired *at the same time* within a single **`locks`** section:

```
var a, b: TLock[2]
var x: TLock[1]
# invalid locking order: TLock[1] cannot be acquired before TLock[2]:
{.locks: [x].}:
  {.locks: [a].}:
    ...
# valid locking order: TLock[2] acquired before TLock[1]:
{.locks: [a].}:
  {.locks: [x].}:
    ...

# invalid locking order: TLock[2] acquired before TLock[2]:
{.locks: [a].}:
  {.locks: [b].}:
    ...

# valid locking order, locks of the same level acquired at the same time:
{.locks: [a, b].}:
  ...
```

Here is how a typical multilock statement can be implemented in Nim. Note how the runtime check is required to ensure a global ordering for two locks **`a`** and **`b`** of the same lock level:

```
template multilock(a, b: ptr TLock; body: untyped) =
  if cast[ByteAddress](a) < cast[ByteAddress](b):
    pthread_mutex_lock(a)
    pthread_mutex_lock(b)
  else:
    pthread_mutex_lock(b)
    pthread_mutex_lock(a)
  {.locks: [a, b].}:
    try:
      body
    finally:
      pthread_mutex_unlock(a)
      pthread_mutex_unlock(b)
```

Whole routines can also be annotated with a **`locks`** pragma that takes a lock level. This then means that the routine may acquire locks of up to this level. This is essential so that procs can be called within a **`locks`** section:

```
proc p() {.locks: 3.} = discard

var a: TLock[4]
{.locks: [a].}:
  # p's locklevel (3) is strictly less than a's (4) so the call is allowed:
  p()
```

As usual **`locks`** is an inferred effect and there is a subtype relation: **`proc () {.locks: N.}`** is a subtype of **`proc () {.locks: M.}`** iff (M <= N).

The **locks** pragma can also take the special value **"unknown"**. This is useful in the context of dynamic method dispatching. In the following example, the compiler can infer a lock level of 0 for the **base** case. However, one of the overloaded methods calls a procvar which is potentially locking. Thus, the lock level of calling **g.testMethod** cannot be inferred statically, leading to compiler warnings. By using **{.locks: "unknown".}**, the base method can be marked explicitly as having unknown lock level as well:

```
type SomeBase* = ref object of RootObj
type SomeDerived* = ref object of SomeBase
  memberProc*: proc ()

method testMethod(g: SomeBase) {.base, locks: "unknown".} = discard
method testMethod(g: SomeDerived) =
  if g.memberProc != nil:
    g.memberProc()
```

Taint mode

The Nim compiler and most parts of the standard library support a taint mode. Input strings are declared with the TaintedString string type declared in the **`system`** module.

If the taint mode is turned on (via the **`--taintMode:on`** command line option) it is a distinct string type which helps to detect input validation errors:

```
echo "your name: "
var name: TaintedString = stdin.readline
# it is safe here to output the name without any input validation, so
# we simply convert `name` to string to make the compiler happy:
echo "hi, ", name.string
```

If the taint mode is turned off, **`TaintedString`** is simply an alias for **`string`**.

Made with Nim. Generated: 2017-09-07 08:48:15 UTC