Nimrod Manual 0.8.15

Andreas Rumpf

May 9, 2012

Contents

"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

1 About this document

Note: This document is a draft! Several of Nimrod's features need more precise wording. This manual will evolve into a proper specification some day.

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

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*; an alternative spelling for optional parts is `[a]`. The `|` symbol is used to mark alternatives and has the lowest precedence. Parentheses may be used to group elements. 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]
```

Other parts of Nimrod - like scoping rules or runtime semantics are only described in an informal manner. The reason is that formal semantics are difficult to write and understand. However, there is only one Nimrod implementation, so one may consider it as the formal specification; especially since the compiler's code is pretty clean (well, some parts of it).

2 Definitions

A Nimrod 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.

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*. However, the implementation provides a means to disable these runtime checks. See the section pragmas7 for details.

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.

3 Lexical Analysis

3.1 Encoding

All Nimrod 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.

3.2 Indentation

Nimrod'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 terminals `IND` (indentation), `DED` (dedentation) and `SAD` (same indentation) are generated by the scanner, denoting an indentation.

These terminals are only generated for lines that are not empty.

The parser and the scanner communicate over a stack which indentation terminal should be generated: the stack consists of integers counting the spaces. The stack is initialized with a zero on its top. The scanner reads from the stack: If the current indentation token consists of more spaces than the entry at the top of the stack, a `IND` token is generated, else if it consists of the same number of spaces, a `SAD` token is generated. If it consists of fewer spaces, a `DED` token is generated for any item on the stack that is greater than the current. These items are later popped from the stack by the parser. At the end of the file, a `DED` token is generated for each number remaining on the stack that is larger than zero.

Because the grammar contains some optional `IND` tokens, the scanner cannot push new indentation levels. This has to be done by the parser. The symbol `indPush` indicates that an `IND` token is expected; the current number of leading spaces is pushed onto the stack by the parser. The symbol `indPop` denotes that the parser pops an item from the indentation stack. No token is consumed by `indPop`.

3.3 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 which is aligned to the preceding one, it does not start a new comment:

```
i = 0     # This is a single comment over multiple lines belonging to the
          # assignment statement. The scanner merges these two pieces.
# This is a new comment belonging to the current block, but to no particular
# statement.
i = i + 1 # This a new comment that is NOT
echo(i)   # continued here, because this comment refers to the echo statement
```

Comments are tokens; they are only allowed at certain places in the input file as they belong to the syntax tree! This feature enables perfect source-to-source transformations (such as pretty-printing) and superior documentation generators. A nice side-effect is that the human reader of the code always knows exactly which code snippet the comment refers to.

3.4 Identifiers & Keywords

Identifiers in Nimrod 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) )*
```

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

```
addr and as asm atomic
bind block break
case cast const continue converter
discard distinct div do
elif else end enum except export
finally for from
generic
if import in include is isnot iterator
lambda let
macro method mod
nil not notin
object of or out
proc ptr
```

	Meaning
	newline
	carriage return
	line feed
	form feed
	tabulator
	vertical tabulator
	backslash
	quotation mark
	apostrophe
	character with decimal value d; all decimal digits directly following are used for the character
	alert
	backspace
	escape [ESC]
	character with hex value HH; exactly two hex digits are allowed

```
raise ref return
shl shr static
template try tuple type
var
when while with without
xor
yield
```

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

Nimrod is a style-insensitive language. This means that it is not case-sensitive and even underscores are ignored: **type** is a reserved word, and so is **TYPE** or **T__Y__P__E**. The idea behind this is that this allows programmers to use their own preferred spelling style and libraries written by different programmers cannot use incompatible conventions. A Nimrod-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.

3.5 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:

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

3.6 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 immediately followed by a newline, the newline 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"
```

3.7 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 tab
```

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.

3.8 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 Nimrod (for example regular expressions).

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

3.9 Character literals

Character literals are enclosed in single quotes '' and can contain the same escape sequences as strings - with one exception: `\n` is not allowed as it may be wider than one character (often it is the pair CR/LF for example). 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 Nimrod can thus support `array[char, int]` or `set[char]` efficiently as many algorithms rely on this feature.

3.10 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'
INT_LIT ::= digit ( ['_'] digit )*
          | '0' ('x' | 'X' ) hexdigit ( ['_'] hexdigit )*
          | '0o' octdigit ( ['_'] octdigit )*
          | '0' ('b' | 'B' ) bindigit ( ['_'] bindigit )*

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'

exponent ::= ('e' | 'E' ) ['+' | '-'] digit ( ['_'] digit )*
FLOAT_LIT ::= digit (['_'] digit)*  ('.' (['_'] digit)* [exponent] |exponent)
FLOAT32_LIT ::= ( FLOAT_LIT | INT_LIT ) '\'' ('f' | 'F') '32'
FLOAT64_LIT ::= ( FLOAT_LIT | INT_LIT ) '\'' ('f' | 'F') '64'
```

	Resulting type of literal
	int8
	int16
	int32
	int64
	float32
	float64

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`) 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`.

The type suffixes are:

Floating point literals may also be in binary, octal or hexadecimal notation: `0B0_10001110100_0000101001000111110` is approximately 1.72826e35 according to the IEEE floating point standard.

3.11 Operators

In Nimrod one can define his own 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 the two tokens * and : (to support `var v*: T`).

3.12 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 {., .}.

4 Syntax

This section lists Nimrod's standard syntax in ENBF. How the parser receives indentation tokens is already described in the Lexical Analysis3 section.

Nimrod allows user-definable operators. Binary operators have 10 different levels of precedence.

4.1 Relevant character

An operator symbol's *relevant character* is its first character unless the first character is \ and its length is greater than 1 then it is the second character.

This rule allows to escape operator symbols with \ and keeps the operator's precedence and associativity; this is useful for meta programming.

4.2 Associativity

All binary operators are left-associative, except binary operators whose relevant char is ^.

	Operators	Relevant character	Terminal symbol
		`$ ^`	OP9
	`* / div mod shl shr %`	`* % \ /`	OP8
	`+ -`	`+ ~ |`	OP7
	`&`	`&`	OP6
	`..`	`.`	OP5
	`== <= < >= > != in not_in is isnot not of`	`= < > !`	OP4
	`and`		OP3
	`or xor`		OP2
		`@ : ?`	OP1
	assignment operator (like +=, *=)		OP0

4.3 Precedence

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

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

If the operator's relevant 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)`.

Otherwise precedence is determined by the relevant character.

The grammar's start symbol is `module`.

```
module ::= ([COMMENT] [SAD] stmt)*

comma ::= ',' [COMMENT] [IND]
operator ::=  OP0 | OP1 | OP2 | OP3 | OP4 | OP5 | OP6 | OP7 | OP8 | OP9
           | 'or' | 'xor' | 'and'
           | 'is' | 'isnot' | 'in' | 'notin' | 'of'
           | 'div' | 'mod' | 'shl' | 'shr' | 'not' | 'addr' | 'static' | '..'

prefixOperator ::= operator

optInd ::= [COMMENT] [IND]
optPar ::= [IND] | [SAD]

lowestExpr ::= assignExpr (OP0 optInd assignExpr)*
assignExpr ::= orExpr (OP1 optInd orExpr)*
orExpr ::= andExpr (OP2 optInd andExpr)*
andExpr ::= cmpExpr (OP3 optInd cmpExpr)*
cmpExpr ::= sliceExpr (OP4 optInd sliceExpr)*
sliceExpr ::= ampExpr (OP5 optInd ampExpr)*
ampExpr ::= plusExpr (OP6 optInd plusExpr)*
plusExpr ::= mulExpr (OP7 optInd mulExpr)*
mulExpr ::= dollarExpr (OP8 optInd dollarExpr)*
dollarExpr ::= primary (OP9 optInd primary)*

indexExpr ::= expr

castExpr ::= 'cast' '[' optInd typeDesc optPar ']' '(' optInd expr optPar ')'
symbol ::= '`' (KEYWORD | IDENT | operator | '(' ')' | '[' ']' | '{' '}'
               | '=' | literal)+ '`'
         | IDENT

primaryPrefix ::= (prefixOperator | 'bind') optInd
primarySuffix ::= '.' optInd symbol [generalizedLit]
                | '(' optInd namedExprList optPar ')'
                | '[' optInd [indexExpr (comma indexExpr)* [comma]] optPar ']'
                | '{' optInd [indexExpr (comma indexExpr)* [comma]] optPar '}'

primary ::= primaryPrefix* (symbol [generalizedLit] |
```

```
                            constructor | castExpr)
            primarySuffix*

generalizedLit ::= GENERALIZED_STR_LIT | GENERALIZED_TRIPLESTR_LIT

literal ::= INT_LIT | INT8_LIT | INT16_LIT | INT32_LIT | INT64_LIT
          | FLOAT_LIT | FLOAT32_LIT | FLOAT64_LIT
          | STR_LIT | RSTR_LIT | TRIPLESTR_LIT
          | CHAR_LIT
          | NIL

constructor ::= literal
          | '[' optInd colonExprList optPar ']'
          | '{' optInd ':' | colonExprList optPar '}'
          | '(' optInd colonExprList optPar ')'

colonExpr ::= expr [':' expr]
colonExprList ::= [colonExpr (comma colonExpr)* [comma]]

namedExpr ::= expr ['=' expr]
namedExprList ::= [namedExpr (comma namedExpr)* [comma]]

exprOrType ::= lowestExpr
            | 'if' expr ':' expr ('elif' expr ':' expr)* 'else' ':' expr
            | 'var' exprOrType
            | 'ref' exprOrType
            | 'ptr' exprOrType
            | 'type' exprOrType
            | 'tuple' tupleDesc

expr ::= exprOrType
     | 'proc' paramList [pragma] ['=' stmt]

exprList ::= [expr (comma expr)* [comma]]


qualifiedIdent ::= symbol ['.' symbol]

typeDesc ::= exprOrType
         | 'proc' paramList [pragma]

macroStmt ::= ':' [stmt] ('of' [exprList] ':' stmt
                         |'elif' expr ':' stmt
                         |'except' exceptList ':' stmt )*
                         ['else' ':' stmt]

pragmaBlock ::= pragma [':' stmt]

simpleStmt ::= returnStmt
             | yieldStmt
             | discardStmt
             | raiseStmt
             | breakStmt
             | continueStmt
             | pragmaBlock
             | importStmt
             | fromStmt
             | includeStmt
             | exprStmt
complexStmt ::= ifStmt | whileStmt | caseStmt | tryStmt | forStmt
              | blockStmt | staticStmt | asmStmt
              | procDecl | iteratorDecl | macroDecl | templateDecl | methodDecl
              | constSection | letSection | varSection
              | typeSection | whenStmt | bindStmt

indPush ::= IND # and push indentation onto the stack
indPop ::= # pop indentation from the stack

stmt ::= simpleStmt [SAD]
 | indPush (complexStmt | simpleStmt)
```

```
  ([SAD] (complexStmt | simpleStmt))*
   DED indPop

exprStmt ::= lowestExpr ['=' expr | [expr (comma expr)*] [macroStmt]]
returnStmt ::= 'return' [expr]
yieldStmt ::= 'yield' expr
discardStmt ::= 'discard' expr
raiseStmt ::= 'raise' [expr]
breakStmt ::= 'break' [symbol]
continueStmt ::= 'continue'
ifStmt ::= 'if' expr ':' stmt ('elif' expr ':' stmt)* ['else' ':' stmt]
whenStmt ::= 'when' expr ':' stmt ('elif' expr ':' stmt)* ['else' ':' stmt]
caseStmt ::= 'case' expr [':'] ('of' exprList ':' stmt)*
                               ('elif' expr ':' stmt)*
                               ['else' ':' stmt]
whileStmt ::= 'while' expr ':' stmt
forStmt ::= 'for' symbol (comma symbol)* 'in' expr ':' stmt
exceptList ::= [qualifiedIdent (comma qualifiedIdent)*]

tryStmt ::= 'try' ':' stmt
           ('except' exceptList ':' stmt)*
           ['finally' ':' stmt]
asmStmt ::= 'asm' [pragma] (STR_LIT | RSTR_LIT | TRIPLESTR_LIT)
blockStmt ::= 'block' [symbol] ':' stmt
staticStmt ::= 'static' ':' stmt
filename ::= symbol | STR_LIT | RSTR_LIT | TRIPLESTR_LIT
importStmt ::= 'import' filename (comma filename)*
includeStmt ::= 'include' filename (comma filename)*
bindStmt ::= 'bind' IDENT (comma IDENT)*
fromStmt ::= 'from' filename 'import' symbol (comma symbol)*

pragma ::= '{.' optInd (colonExpr [comma])* optPar ('.}' | '}')

param ::= symbol (comma symbol)* (':' typeDesc ['=' expr] | '=' expr)
paramList ::= ['(' [param (comma param)*] optPar ')'] [':' typeDesc]

genericConstraint ::= 'object' | 'tuple' | 'enum' | 'proc' | 'ref' | 'ptr'
                    | 'var' | 'distinct' | primary
genericConstraints ::= genericConstraint ( '|' optInd genericConstraint )*

genericParam ::= symbol [':' genericConstraints] ['=' expr]
genericParams ::= '[' genericParam (comma genericParam)* optPar ']'


routineDecl := symbol ['*'] [genericParams] paramList [pragma] ['=' stmt]
procDecl ::= 'proc' routineDecl
macroDecl ::= 'macro' routineDecl
iteratorDecl ::= 'iterator' routineDecl
templateDecl ::= 'template' routineDecl
methodDecl ::= 'method' routineDecl

colonAndEquals ::= [':' typeDesc] '=' expr

constDecl ::= symbol ['*'] [pragma] colonAndEquals [COMMENT | IND COMMENT]
            | COMMENT
constSection ::= 'const' indPush constDecl (SAD constDecl)* DED indPop
letSection ::= 'let' indPush constDecl (SAD constDecl)* DED indPop

typeDef ::= typeDesc | objectDef | enumDef | 'distinct' typeDesc

objectField ::= symbol ['*'] [pragma]
objectIdentPart ::= objectField (comma objectField)* ':' typeDesc
                    [COMMENT|IND COMMENT]

objectWhen ::= 'when' expr ':' [COMMENT] objectPart
              ('elif' expr ':' [COMMENT] objectPart)*
              ['else' ':' [COMMENT] objectPart]
objectCase ::= 'case' expr ':' typeDesc [COMMENT]
              ('of' exprList ':' [COMMENT] objectPart)*
              ['else' ':' [COMMENT] objectPart]
```

```
objectPart ::= objectWhen | objectCase | objectIdentPart | 'nil'
             | indPush objectPart (SAD objectPart)* DED indPop
tupleDesc ::= '[' optInd [param (comma param)*] optPar ']'

objectDef ::= 'object' [pragma] ['of' typeDesc] objectPart
enumField ::= symbol ['=' expr]
enumDef ::= 'enum' (enumField [comma] [COMMENT | IND COMMENT])+

typeDecl ::= COMMENT
           | symbol ['*'] [genericParams] ['=' typeDef] [COMMENT | IND COMMENT]

typeSection ::= 'type' indPush typeDecl (SAD typeDecl)* DED indPop

colonOrEquals ::= ':' typeDesc ['=' expr] | '=' expr
varField ::= symbol ['*'] [pragma]
varPart ::= symbol (comma symbol)* colonOrEquals [COMMENT | IND COMMENT]
varSection ::= 'var' (varPart
                  | indPush (COMMENT|varPart)
                    (SAD (COMMENT|varPart))* DED indPop)
```

5 Semantics

5.1 Types

All expressions have a type which is known at compile time. Nimrod 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

5.1.1 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.

	meaning
	unsigned integer addition
	unsigned integer subtraction
	unsigned integer multiplication
	unsigned integer division
	unsigned integer modulo operation
	treat `a` and `b` as unsigned and compare
	treat `a` and `b` as unsigned and compare
	extends the bits of `a` with zeros until it has the width of the `int` type
	treats `a` as unsigned and converts it to an unsigned integer of 8 bits (but still the `int8` type)
	treats `a` as unsigned and converts it to an unsigned integer of 16 bits (but still the `int16` type)
	treats `a` as unsigned and converts it to an unsigned integer of 32 bits (but still the `int32` type)

5.1.2 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.

There are no unsigned integer types, only unsigned operations that treat their arguments as unsigned. Unsigned operations all wrap around; they cannot lead to over- or underflow errors. Unsigned operations use the `%` suffix as convention:

Automatic type conversion is performed in expressions where different kinds of integer types are used: the smaller type is converted to the larger. For further details, see Convertible relation5.2.4.

5.1.3 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 relation5.2.4 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 Nimrod exceptions: EFloatInvalidOp, EFloatDivByZero, EFloatOverflow, EFloatUnderflow, and EFloatInexact. These exceptions inherit from the EFloatingPoint base class.

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

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

In the current implementation `EFloatDivByZero` and `EFloatInexact` are never raised. `EFloatOverflow` is raised instead of `EFloatDivByZero`. 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.

5.1.4 Boolean type

The boolean type is named bool in Nimrod 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.

5.1.5 Character type

The character type is named `char` in Nimrod. 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 Nimrod can support `array[char, int]` or `set[char]` efficiently as many algorithms rely on this feature. The *TRune* type is used for Unicode characters, it can represent any Unicode character. `TRune` is declared in the `unicode` module.

5.1.6 Enumeration types

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

```
type
  TDirection = 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
  TTokenType = 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 explicitely giving the string values to use:

```
type
  TMyEnum = 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.

5.1.7 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
  TSubrange = range[0..5]
```

`TSubrange` is a subrange of an integer which can only hold the values 0 to 5. Assigning any other value to a variable of type `TSubrange` 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).

5.1.8 String type

All string literals are of the type string. A string in Nimrod is very similar to a sequence of characters. However, strings in Nimrod 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.

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 can be used for iteration over all Unicode characters.

5.1.9 CString type

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 Nimrod `string` is implicitely convertible to `cstring` for convenience. If a Nimrod string is passed to a C-style variadic proc, it is implicitely 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 implict, 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.

5.1.10 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.

5.1.11 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 `[]`.

Sequences are similar to arrays but of dynamic length which may change during runtime (like strings). 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
  TIntArray = array[0..5, int] # an array that is indexed with 0..5
  TIntSeq = seq[int] # a sequence of integers
var
  x: TIntArray
  y: TIntSeq
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
```

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.

An open array is also a means to implement passing a variable number of arguments to a procedure. The compiler converts the list of arguments to an array automatically:

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

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

This transformation is only done if the openarray parameter is the last parameter in the procedure header. The current implementation does not support nested open arrays.

5.1.12 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 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 may change in future versions of the compiler.

```
type
  TPerson = tuple[name: string, age: int] # type representing a person
                                          # a person consists of a name
                                          # and an age
var
  person: TPerson
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.

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.

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

  TStudent = object of TPerson # a student is a person
    id: int                    # with an id field

var
  student: TStudent
  person: TPerson
assert(student of TStudent) # is 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*.

5.1.13 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 Nimrod
type
  TNodeKind = 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
  PNode = ref TNode
  TNode = object
    case kind: TNodeKind  # the ''kind'' field is the discriminator
    of nkInt: intVal: int
    of nkFloat: floatVal: float
```

	meaning
	union of two sets
	intersection of two sets
	difference of two sets (A without B's elements)
	set equality
	subset relation (A is subset of B or equal to B)
	strong subset relation (A is a real subset of B)
	set membership (A contains element e)
	symmetric set difference (= (A - B) + (B - A))
	the cardinality of A (number of elements in A)
	same as A = A + {elem}
	same as A = A - {elem}

```
    of nkString: strVal: string
    of nkAdd, nkSub:
      leftOp, rightOp: PNode
    of nkIf:
      condition, thenPart, elsePart: PNode

var
  n: PNode
new(n)  # creates a new node
n.kind = nkFloat
n.floatVal = 0.0 # valid, because ``n.kind==nkFloat``, so that it fits

# the following statement raises an `EInvalidField` exception, because
# n.kind's value does not fit:
n.strVal = ""
```

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.

5.1.14 Set type

The set type models the mathematical notion of a set. The set's basetype can only be an ordinal type. The reason is that sets are implemented as high performance bit vectors.

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

```
{'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:

5.1.15 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).

Nimrod 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.

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
  PNode = ref TNode
  TNode = object
    le, ri: PNode
    data: int

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

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
  TData = tuple[x, y: int, s: string]

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

# 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 TData`. 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. You need to know low level details like this when mixing garbage collected data with unmanaged memory.

5.1.16 Procedural type

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

Examples:

```
type
  TCallback = proc (x: int) {.cdecl.}

proc printItem(x: Int) = ...

proc forEach(c: TCallback) =
  ...

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

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

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

proc setOnMouseMove(mouseMoveEvent: TOnMouseMove) = nil

# 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`.

Nimrod supports these calling conventions:

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 Nimrod 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.

nimcall Nimcall is the default convention used for Nimrod procedures. It is the same as `fastcall`, but only for C compilers that support `fastcall`.

closure 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 implicitely passed environment.

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 Nimrod's default calling convention for procedures is `fastcall` to improve speed.

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

Assigning/passing a procedure to a procedural variable is only allowed if one of the following conditions hold:

1. The procedure that is accessed resists in the current module.
2. The procedure is marked with the `procvar` pragma (see procvar pragma7.2).
3. The procedure has a calling convention that differs from `nimcall`.
4. The procedure is anonymous.

The rules' purpose is to prevent the case that extending a non-`procvar` procedure with default parameters breaks client code.

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 wether it accesses its environment. If it does so, it has the calling convention `closure`, otherwise it has the calling convention `nimcall`.

5.1.17 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.

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
  TDollar = distinct int
  TEuro = distinct int

var
  d: TDollar
  e: TEuro

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

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

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

proc `*` (x: int, y: TDollar): TDollar =
  result = TDollar(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: TDollar, y: int): TDollar {.borrow.}
proc `*` (x: int, y: TDollar): TDollar {.borrow.}
proc `div` (x: TDollar, y: int): TDollar {.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 `TEuro` currency. This can be solved with templates5.5.

```
template Additive(typ: typeDesc): stmt =
  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): stmt =
  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): stmt =
  proc `<` * (x, y: typ): bool {.borrow.}
  proc `<=` * (x, y: typ): bool {.borrow.}
  proc `==` * (x, y: typ): bool {.borrow.}

template DefineCurrency(typ, base: expr): stmt =
  type
    typ* = distinct base
  Additive(typ)
  Multiplicative(typ, base)
  Comparable(typ)

DefineCurrency(TDollar, int)
DefineCurrency(TEuro, int)
```

5.1.18 Void type

The void type denotes the absense 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) = nil
proc emptyProc() = nil

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`.

5.2 Type relations

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

5.2.1 Type equality

Nimrod 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 set[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: set[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.

5.2.2 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)
```

5.2.3 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)
```

5.2.4 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 =
  case a.kind
  of int8:    result = b.kind in {int16, int32, int64, int}
  of int16:   result = b.kind in {int32, int64, int}
  of int32:   result = b.kind in {int64, int}
  of float:   result = b.kind in {float32, float64}
  of float32: result = b.kind in {float64, float}
  of float64: result = b.kind in {float32, float}
  of seq:
    result = b.kind == openArray and typeEquals(a.baseType, b.baseType)
  of array:
    result = b.kind == openArray and typeEquals(a.baseType, b.baseType)
    if a.baseType == char and a.indexType.rangeA == 0:
      result = b.kind = cstring
  of cstring, ptr:
    result = b.kind == pointer
  of string:
    result = b.kind == 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 =
  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
  return false
```

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.

5.2.5 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.

5.2.6 Overloading resolution

To be written.

5.3 Statements and expressions

Nimrod uses the common statement/expression paradigm: Statements do not produce a value in contrast to expressions. Call expressions are statements. If the called procedure returns a value, it is not a valid statement as statements do not produce values. To evaluate an expression for side-effects and throw its value away, one can use the `discard` statement.

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 intended:

```
simpleStmt ::= returnStmt
           | yieldStmt
           | discardStmt
           | raiseStmt
           | breakStmt
           | continueStmt
           | pragma
           | importStmt
           | fromStmt
           | includeStmt
           | exprStmt
complexStmt ::= ifStmt | whileStmt | caseStmt | tryStmt | forStmt
              | blockStmt | asmStmt
              | procDecl | iteratorDecl | macroDecl | templateDecl
              | constSection | letSection
              | typeSection | whenStmt | varSection
```

5.3.1 Discard statement

Syntax:

```
discardStmt ::= 'discard' expr
```

Example:

```
proc p(x, y: int): int =
  return 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 implicitely if the called proc/iterator has been declared with the discardable pragma:

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

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

	default value
	0
	0.0
	'\0'
	false
	nil
	nil
	nil (*not* `@[]`)
	nil (*not* `""`)
	(default(A), default(B), ...) (analogous for objects)
	[default(T), ...]
	default(T); this may be out of the valid range
	cast[T](0); this may be an invalid value

5.3.2 Var statement

Syntax:

```
colonOrEquals ::= ':' typeDesc ['=' expr] | '=' expr
varField ::= symbol ['*'] [pragma]
varPart ::= symbol (comma symbol)* [comma] colonOrEquals [COMMENT | IND COMMENT]
varSection ::= 'var' (varPart
                   | indPush (COMMENT|varPart)
                     (SAD (COMMENT|varPart))* DED indPop)
```

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.

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.} = nil
```

5.3.3 let statement

A Let statement declares new local and global single assignment variables and binds a value to them. The syntax is the 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.

5.3.4 Const section

Syntax:

```
colonAndEquals ::= [':' typeDesc] '=' expr

constDecl ::= symbol ['*'] [pragma] colonAndEquals [COMMENT | IND COMMENT]
            | COMMENT
constSection ::= 'const' indPush constDecl (SAD constDecl)* DED indPop
```

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.

Nimrod 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 pragma7.1 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.

5.3.5 Static statement/expression

Syntax:

```
staticExpr ::= 'static' '(' optInd expr optPar ')'
staticStmt ::= 'static' ':' stmt
```

A static statement/expression can be used to enforce compile time evaluation explicitely. 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 Nimrod will support the FFI at compile time.

5.3.6 If statement

Syntax:

```
ifStmt ::= 'if' expr ':' stmt ('elif' expr ':' stmt)* ['else' ':' stmt]
```

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 statement after the `if` statement.

5.3.7 Case statement

Syntax:

```
caseStmt ::= 'case' expr [':'] ('of' sliceExprList ':' stmt)*
                               ('elif' expr ':' stmt)*
                               ['else' ':' stmt]
```

Example:

```
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. If the expression is not of an ordinal type, and no `else` part is given, control passes after the `case` statement.

To suppress the static error in the ordinal case an `else` part with a `nil` statement can be used.

As a special semantic extension, an expression in an `of` branch of a case statement may evaluate to a set constructor; the set 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"
```

5.3.8 When statement

Syntax:

```
whenStmt ::= 'when' expr ':' stmt ('elif' expr ':' stmt)* ['else' ':' stmt]
```

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 `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 `expr` 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.

5.3.9 Raise statement

Syntax:

```
raiseStmt ::= 'raise' [expr]
```

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 ENoExceptionToReraise exception is raised if there is no exception to re-raise. It follows that the `raise` statement *always* raises an exception.

5.3.10 Try statement

Syntax:

```
qualifiedIdent ::= symbol ['.' symbol]
exceptList ::= [qualifiedIdent (comma qualifiedIdent)* [comma]]
tryStmt ::= 'try' ':' stmt
           ('except' exceptList ':' stmt)*
           ['finally' ':' stmt]
```

Example:

```
# read the first two lines of a text file that should contain numbers
# and tries to add them
var
  f: TFile
if open(f, "numbers.txt"):
  try:
    var a = readLine(f)
    var b = readLine(f)
    echo("sum: " & $(parseInt(a) + parseInt(b)))
  except EOverflow:
    echo("overflow!")
  except EInvalidValue:
    echo("could not convert string to integer")
  except EIO:
    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 of the list `exceptlist` 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 in no list. 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).

5.3.11 Return statement

Syntax:

```
returnStmt ::= 'return' [expr]
```

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
```

5.3.12 Yield statement

Syntax:

```
yieldStmt ::= 'yield' expr
```

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 statement5.3.28) for further information.

5.3.13 Block statement

Syntax:

```
blockStmt ::= 'block' [symbol] ':' stmt
```

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.

5.3.14 Break statement

Syntax:

```
breakStmt ::= 'break' [symbol]
```

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.

5.3.15 While statement

Syntax:

```
whileStmt ::= 'while' expr ':' stmt
```

Example:

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

5.3.16 Continue statement

Syntax:

```
continueStmt ::= 'continue'
```

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
```

5.3.17 Assembler statement

Syntax:

```
asmStmt ::= 'asm' [pragma] (STR_LIT | RSTR_LIT | TRIPLESTR_LIT)
```

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

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

5.3.18 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 (but unlikely to be good style).

5.3.19 Table constructor

A table constructor is syntactic sugar for an array constructor:

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

# is the same as:
[("key1", "value1"), ("key2", "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 unusal 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.

5.3.20 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).

5.3.21 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.

5.3.22 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`. 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.

5.3.23 Procedures

What most programming languages call methods or functions are called procedures in Nimrod (which is the correct terminology). A procedure declaration defines an identifier and associates it with a block of code. A procedure may call itself recursively. A parameter may be given a default value that is used if the caller does not provide a value for this parameter. The syntax is:

```
param ::= symbol (comma symbol)* (':' typeDesc ['=' expr] | '=' expr)
paramList ::= ['(' [param (comma param)*] [SAD] ')'] [':' typeDesc]

genericParam ::= symbol [':' typeDesc] ['=' expr]
genericParams ::= '[' genericParam (comma genericParam)* [SAD] ']'

procDecl ::= 'proc' symbol ['*'] [genericParams] paramList [pragma]
             ['=' stmt]
```

If the = `stmt` part is missing, 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 tries to find the proc that 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'
```

A procedure cannot modify its parameters (unless the parameters have the type *var*).

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.
  return 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
  return a * b + c

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

5.3.24 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] =
  return (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
```

5.3.25 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 implicitely 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.

5.3.26 Overloading of the subscript operator

The `[]` subscript operator for arrays/openarrays/sequences can be overloaded. Overloading support is only possible if the first parameter has no type that already supports the built-in `[]` notation. Currently the compiler does not check this restriction.

5.3.27 Multi-methods

Procedures always use static dispatch. Multi-methods use dynamic dispatch.

```
type
  TExpr = object ## abstract base class for an expression
  TLiteral = object of TExpr
    x: int
  TPlusExpr = object of TExpr
    a, b: ref TExpr

method eval(e: ref TExpr): int =
  # override this base method
  quit "to override!"

method eval(e: ref TLiteral): int = return e.x

method eval(e: ref TPlusExpr): int =
  # watch out: relies on dynamic binding
  return eval(e.a) + eval(e.b)

proc newLit(x: int): ref TLiteral =
  new(result)
  result.x = x

proc newPlus(a, b: ref TExpr): ref TPlusExpr =
  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.

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

```
type
  TThing = object
  TUnit = object of TThing
    x: int

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

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

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

var
  a, b: TUnit
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 `TUnit, TThing` is prefered over `TThing, TUnit`.

Performance note: Nimrod 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.

5.3.28 Iterators and the for statement

Syntax:

```
forStmt ::= 'for' symbol (comma symbol)* [comma] 'in' expr ':' stmt

param ::= symbol (comma symbol)* [comma] ':' typeDesc
```

```
paramList ::= ['(' [param (comma param)* [comma]] ')'] [':' typeDesc]

genericParam ::= symbol [':' typeDesc]
genericParams ::= '[' genericParam (comma genericParam)* [comma] ']'

iteratorDecl ::= 'iterator' symbol ['*'] [genericParams] paramList [pragma]
             ['=' stmt]
```

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 (`x` in the example) - 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 is always 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)
```

The current implementation always inlines the iterator code leading to zero overhead for the abstraction. But this may increase the code size. Later versions of the compiler will only inline iterators which have the calling convention `inline`.

If the iterator yields a tuple, there have to 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.

Implict items/pairs invokations 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 implicitely invoked:

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

If the for loop has exactly 2 variables, a `pairs` iterator is implicitely invoked.

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

5.3.29 Type sections

Syntax:

```
typeDef ::= typeDesc | objectDef | enumDef

genericParam ::= symbol [':' typeDesc]
genericParams ::= '[' genericParam (comma genericParam)* [comma] ']'

typeDecl ::= COMMENT
           | symbol ['*'] [genericParams] ['=' typeDef] [COMMENT|IND COMMENT]

typeSection ::= 'type' indPush typeDecl (SAD typeDecl)* DED indPop
```

Example:

```
type # example demonstrates mutually recursive types
  PNode = ref TNode # a traced pointer to a TNode
  TNode = object
    le, ri: PNode   # left and right subtrees
    sym: ref TSym   # leaves contain a reference to a TSym

  TSym = object     # a symbol
    name: string    # the symbol's name
    line: int       # the line the symbol was declared in
    code: PNode     # 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.

5.4 Generics

Example:

```
type
  TBinaryTree[T] = object      # TBinaryTree is a generic type with
                               # with generic param ''T''
    le, ri: ref TBinaryTree[T] # left and right subtrees; may be nil
    data: T                    # the data stored in a node
  PBinaryTree[T] = ref TBinaryTree[T] # a shorthand for notational convenience

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

proc add[T](root: var PBinaryTree[T], n: PBinaryTree[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: PBinaryTree[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: PBinaryTree[string] # instantiate a PBinaryTree with the type string
add(root, newNode("hallo")) # instantiates generic procs ''newNode'' and
add(root, newNode("world")) # ''add''
for str in inorder(root):
  writeln(stdout, str)
```

	matches
	any object type
	any tuple type
	any enumeration
	any proc type
	any `ref` type
	any `ptr` type
	any `var` type
	any distinct type
	any array type
	any set type
	any seq type

Generics are Nimrod'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.

5.4.1 Is operator

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

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

5.4.2 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)
```

5.4.3 Type constraints

Type constraints can be used to restrict the instantiation of a generic type parameter. Only the specified types are valid for instantiation:

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

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

Apart from ordinary types, type constraints can also be of the following *type classes*:
The following example is taken directly from the system module:

```
proc `==`*[T: tuple](x, y: T): bool =
  ## generic ``==`` operator for tuples that is lifted from the components
  ## of `x` and `y`.
  for a, b in fields(x, y):
    if a != b: return false
  return true
```

5.4.4 Symbol lookup in generics

Symbols in generics are looked up in two different contexts: Both the context at definition and the context at instantiation are considered for any symbol occuring in a generic:

```
type
  TIndex = distinct int

proc `==` (a, b: TIndex): bool {.borrow.}

var a = (0, 0.TIndex)
var b = (0, 0.TIndex)

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 `TIndex` type is defined *after* the == for tuples; yet the example compiles as the instantiation takes the currently defined symbols into account too.

5.5 Templates

A template is a simple form of a macro: It is a simple substitution mechanism that operates on Nimrod'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: expr): expr =
  # 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 `expr` (stands for *expression*), `stmt` (stands for *statement*) or `typedesc` (stands for *type description*). These are no real types, they just help the compiler parsing. Real types can be used too; this implies that expressions are expected. However, for parameter type checking the arguments are semantically checked before being passed to the template. Other arguments are not semantically checked before being passed to the template.

The template body does not open a new scope. To open a new scope a `block` statement can be used:

```
template declareInScope(x: expr, t: typeDesc): stmt =
  var x: t

template declareInNewScope(x: expr, t: typeDesc): stmt =
  # open a new scope:
  block:
    var x: t

declareInScope(a, int)
a = 42  # works, `a` is known here

declareInNewScope(b, int)
b = 42  # does not work, `b` is unknown
```

If there is a `stmt` parameter it should be the last in the template declaration, because statements are passed to a template via a special **:** syntax:

```
template withFile(f, fn, mode: expr, actions: stmt): stmt =
  block:
    var f: TFile
    if open(f, fn, mode):
      try:
        actions
      finally:
        close(f)
    else:
      quit("cannot open: " & fn)

withFile(txt, "ttempl3.txt", fmWrite):
  txt.writeln("line 1")
  txt.writeln("line 2")
```

In the example the two `writeln` statements are bound to the `actions` parameter.

Note: The symbol binding rules for templates might change!

Symbol binding within templates happens after template instantation:

```
# Module A
var
  lastId = 0

template genId*: expr =
  inc(lastId)
  lastId

# Module B
import A

echo genId() # Error: undeclared identifier: 'lastId'
```

5.5.1 Bind statement

Syntax:

```
bindStmt ::= 'bind' IDENT (comma IDENT)*
```

Exporting a template is a often a leaky abstraction as it can depend on symbols that are not visible from a client module. However, to compensate for this case, a bind statement can be used: It declares all identifiers that should be bound early (i.e. when the template is parsed):

```
# Module A
var
  lastId = 0

template genId*: expr =
  bind lastId
  inc(lastId)
  lastId

# Module B
import A

echo genId() # Works
```

A `bind` statement can also be used in generics for the same purpose.

5.5.2 Identifier construction

In templates identifiers can be constructed with the backticks notation:

```
template typedef(name: expr, typ: typeDesc) =
  type
    `T name`* = typ
    `P name`* = ref `T name`

typedef(myint, int)
var x: PMyInt
```

In the example `name` is instantiated with `myint`, so ‘T name‘ becomes `Tmyint`.

5.6 Macros

Macros are the most powerful feature of Nimrod. They can be used to implement domain specific languages.

While macros enable advanced compile-time code transformations, they cannot change Nimrod's syntax. However, this is no real restriction because Nimrod's syntax is flexible enough anyway.

To write macros, one needs to know how the Nimrod 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*)

5.6.1 Expression Macros

The following example implements a powerful `debug` command that accepts a variable number of arguments:

```
# to work with Nimrod syntax trees, we need an API that is defined in the
# ``macros`` module:
import macros

macro debug(n: expr): stmt =
  # `n` is a Nimrod 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 1..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("writeln", 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, ": ")
writeln(stdout, a[0])

write(stdout, "a[1]")
write(stdout, ": ")
writeln(stdout, a[1])

write(stdout, "x")
write(stdout, ": ")
writeln(stdout, x)
```

5.6.2 Statement Macros

Statement macros are defined just as expression macros. However, they are invoked by an expression following a colon:

```
exprStmt ::= lowestExpr ['=' expr | [expr (comma expr)* [comma]] [macroStmt]]
macroStmt ::= ':' [stmt] ('of' [sliceExprList] ':' stmt
                         | 'elif' expr ':' stmt
                         | 'except' exceptList ':' stmt )*
                          ['else' ':' stmt]
```

The following example outlines a macro that generates a lexical analyzer from regular expressions:

```
import macros

macro case_token(n: stmt): stmt =
  # creates a lexical analyzer from regular expressions
  # ... (implementation is an exercise for the reader :-)
  nil

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.

5.7 Modules

Nimrod 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.

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
  return x + 1
```

5.8 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.

5.8.1 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.

5.8.2 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.

5.8.3 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 other 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
```

6 Compiler Messages

The Nimrod compiler emits different kinds of messages: hint, warning, and error messages. An *error* message is emitted if the compiler encounters any static error.

7 Pragmas

Syntax:

```
colonExpr ::= expr [':' expr]
colonExprList ::= [colonExpr (comma colonExpr)* [comma]]

pragma ::= '{.' optInd (colonExpr [comma])* [SAD] ('.}' | '}')
```

Pragmas are Nimrod'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.

7.1 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.

Future directions: `func` may become a keyword and syntactic sugar for a proc with no side effects:

```
func `+` (x, y: int): int
```

7.2 procvar pragma

The procvar pragma is used to mark a proc that it can be passed to a procedural variable.

7.3 compileTime pragma

The compileTime pragma is used to mark a proc to be used at compile time only. No code will be generated for it. Compile time procs are useful as helpers for macros.

7.4 noReturn pragma

The noreturn pragma is used to mark a proc that never returns.

7.5 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
  PNode = ref TNode
  TNode {.acyclic, final.} = object
    left, right: PNode
    data: string
```

In the example a tree structure is declared with the `TNode` 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
  PNode = acyclic ref TNode
  TNode = object
    left, right: PNode
    data: string
```

7.6 Final pragma

The final pragma can be used for an object type to specify that it cannot be inherited from.

7.7 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 Nimrod require deep copying of sequences and strings. This can be expensive, especially if sequences are used to build a tree structure:

```
type
  TNodeKind = enum nkLeaf, nkInner
  TNode {.final, shallow.} = object
    case kind: TNodeKind
    of nkLeaf:
      strVal: string
    of nkInner:
      children: seq[TNode]
```

7.8 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 is necessary for binary compatibility with other compiled languages.

7.9 NoStackFrame pragma

A proc can be marked with the noStackFrame 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. This is useful for procs that only consist of an assembler statement.

7.10 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.}
```

7.11 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.

7.12 warning pragma

The warning pragma is used to make the compiler output a warning message with the given content. Compilation continues after the warning.

7.13 hint pragma

The hint pragma is used to make the compiler output a hint message with the given content. Compilation continues after the hint.

7.14 line pragma

The line pragma can be used to affect line information of the annotated statement as seen in stack backtraces:

```
template myassert*(cond: expr, 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.

7.15 linearScanEnd pragma

The linearScanEnd pragma can be used to tell the compiler how to compile a Nimrod case statement. Syntactially 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.

7.16 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.

7.17 compilation option pragmas

The listed pragmas here can be used to override the code generation options for a section of code.

The implementation currently provides the following possible options (various others may be added later).

Example:

```
{.checks: off, optimization: speed.}
# compile without runtime checks and optimize for speed
```

	allowed values	description
	on|off	Turns the code generation for all runtime checks on or off.
	on|off	Turns the code generation for array bound checks on or off.
	on|off	Turns the code generation for over- or underflow checks on or off.
	on|off	Turns the code generation for nil pointer checks on or off.
	on|off	Turns the code generation for assertions on or off.
	on|off	Turns the warning messages of the compiler on or off.
	on|off	Turns the hint messages of the compiler on or off.
	none|speed|size	Optimize the code for speed or size, or disable optimization.
	cdecl|...	Specifies the default calling convention for all procedures (and procedure types) that follow.

7.18 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
```

7.19 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 an bytecode interpreter for example) it may provide benefits, though.

7.20 global pragma

The *global*:idx 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.

7.21 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 appers 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.}
```

7.22 Pragma pragma

The pragma pragma can be used to declare user defined pragmas. This is useful because Nimrod'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.} =
  return 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.

7.23 Disabling certain messages

Nimrod 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:

```
{.warning[LineTooLong]: off.} # turn off warning about too long lines
```

This is often better than disabling all warnings at once.

8 Foreign function interface

Nimrod's FFI (foreign function interface) is extensive and only the parts that scale to other future backends (like the LLVM/EcmaScript backends) are documented here.

8.1 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 Nimrod identifier *exactly as spelled*:

```
proc printf(formatstr: cstring) {.importc: "printf", varargs.}
```

Note that this pragma is somewhat of a misnomer: Other backends will provide the same feature under the same name.

8.2 Exportc pragma

The exportc pragma provides a means to export a type, a variable, or a procedure to C. The optional argument is a string containing the C identifier. If the argument is missing, the C name is the Nimrod identifier *exactly as spelled*:

```
proc callme(formatstr: cstring) {.exportc: "callMe", varargs.}
```

Note that this pragma is somewhat of a misnomer: Other backends will provide the same feature under the same name.

8.3 Varargs pragma

The varargs pragma can be applied to procedures only (and procedure types). It tells Nimrod that the proc can take a variable number of parameters after the last specified parameter. Nimrod 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
```

8.4 Dynlib pragma for import

With the dynlib pragma a procedure 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.

8.5 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.

9 Threads

Even though Nimrod's thread support and semantics are preliminary, they should be quite usable already. 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 and channels modules for the thread API.

Nimrod'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 will become the norm in the future.

9.1 Thread pragma

A proc that is executed as a new thread of execution should be marked by the thread pragma. The compiler checks procedures marked as `thread` 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.

Since the semantic checking of threads requires whole program analysis, it is quite expensive and can be turned off with `--threadanalysis:off` to improve compile times.

A thread proc is passed to `createThread` and invoked indirectly; so the `thread` pragma implies `procvar`.

9.2 Threadvar pragma

A global variable can be marked with the threadvar pragma; it is a thead-local variable then:

```
var checkpoints* {.threadvar.}: seq[string] = @[]
```

9.3 Actor model

Caution: This section is already outdated! XXX

Nimrod supports the actor model of concurrency natively:

```
type
  TMsgKind = enum
    mLine, mEof
  TMsg = object {.pure, final.}
    case k: TMsgKind
    of mEof: nil
    of mLine: data: string

var
  thr: TThread[TMsg]
  printedLines = 0
  m: TMsg

proc print() {.thread.} =
  while true:
    var x = recv[TMsg]()
    if x.k == mEof: break
    echo x.data
    discard atomicInc(printedLines)

createThread(thr, print)

var input = open("readme.txt")
while not endOfFile(input):
  m.data = input.readLine()
  thr.send(m)
close(input)
m.k = mEof
thr.send(m)
```

```
joinThread(thr)

echo printedLines
```

In the actor model threads communicate only over sending messages (send and recv built-ins), not by sharing memory. Every thread has an inbox that keeps incoming messages until the thread requests a new message via the `recv` operation. The inbox is an unlimited FIFO queue.

In the above example the `print` thread also communicates with its parent thread over the `printedLines` global variable. In general, it is highly advisable to only read from globals, but not to write to them. In fact a write to a global that contains GC'ed memory is always wrong, because it violates the *no heap sharing restriction*:

```
var
  global: string
  t: TThread[string]

proc horrible() {.thread.} =
  global = "string in thread local heap!"

createThread(t, horrible)
joinThread(t)
```

For the above code the compiler produces "Warning: write to foreign heap". This warning might become an error message in future versions of the compiler.

Creating a thread is an expensive operation, because a new stack and heap needs to be created for the thread. It is therefore highly advisable that a thread handles a large amount of work. Nimrod prefers *coarse grained* over *fine grained* concurrency.

9.4 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*!

10 Taint mode

The Nimrod 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`.