NIFC dialect

NIFC is a dialect of NIF designed to be very close to C. Its benefits are:

- Easier to generate than generating C/C++ code directly.
- Has all the NIF related tooling support.
- NIFC improves upon C's quirky array and pointer type confusion by clearly distinguishing between `array` which is always a value type, `ptr` which always points to a single element and `aptr` which points to an array of elements.
- Inheritance is modelled directly in the type system as opposed to C's quirky type aliasing rule that is concerned with aliasings between a struct and its first element.

Name mangling

Name mangling is performed by NIFC. The following assumptions are made:

- A NIF symbol has the form `identifier.<number>.modulesuffix` (if it is a top level entry) or `identifier.<number>` (for a local symbol). For example `replace.2.strutils` would be the 2nd `replace` from `strutils`. Generic instances get a `.g` suffix.
- Symbols that are imported from C or C++ have `.c` as the `modulesuffix`. Note that via `\xx` a name can contain `::` which is required for C++ support. These symbols can have different names in Nim. The original names can be made available via a `was` annotation. See the grammar for further details.

Names ending in `.c` are mangled by removing the `.c` suffix. For other names the `.` is replaced by _ and _ is encoded as `Q_`.

By design names not imported from C contain a digit somewhere and thus cannot conflict with a keyword from C or C++.

Other characters or character combinations (whether they are valid in C/C++ identifiers or not!) are encoded via this table:

Character combination	Encoded as
`Q`	`QQ` (thanks to this rule `Q` is now available as an escape character)
`_`	`Q_`
`[]=`	`putQ`
`[]`	`getQ`
`$`	`dollarQ`
`%`	`percentQ`
`&`	`ampQ`
`^`	`roofQ`
`!`	`emarkQ`

Character combination	Encoded as
`?`	`qmarkQ`
`*`	`starQ`
`+`	`plusQ`
`-`	`minusQ`
`/`	`slashQ`
`\`	`bslashQ`
`==`	`eqQ`
`=`	`eQ`
`<=`	`leQ`
`>=`	`geQ`
`<`	`ltQ`
`>`	`gtQ`
`~`	`tildeQ`
`:`	`colonQ`
`@`	`atQ`
`|`	`barQ`
Other	`XxxQ` where `xx` is the hexadecimal value

Grammar

Generated NIFC code must adhere to this grammar. For better readability `'('` and `')'` are written without quotes and `[]` is used for grouping.

```
Empty ::= <according to NIF's spec>
Identifier ::= <according to NIF's spec>
Symbol ::= <according to NIF's spec>
SymbolDef ::= <according to NIF's spec>
Number ::= <according to NIF's spec>
CharLiteral ::= <according to NIF's spec>
StringLiteral ::= <according to NIF's spec>
IntBits ::= [0-9]+ | 'M'

Lvalue ::= Symbol | (deref Expr) |
             (at Expr Expr) | # array indexing
             (dot Expr Symbol Number) | # field access
             (pat Expr Expr) | # pointer indexing

Call ::= (call Expr+ )
```

```
Expr ::= Number | CharLiteral | StringLiteral |
         Lvalue |
         (par Expr) | # wraps the expression in parentheses
         (addr Lvalue) | # "address of" operation
         (nil) | (false) | (true) |
         (and Expr Expr) | # "&&"
         (or Expr Expr) | # "||"
         (not Expr) | # "!"
         (sizeof Expr) |
         (oconstr Type (kv Symbol Expr)*) |  # (object constructor){...}
         (aconstr Type Expr*) |              # array constructor
         (add Type Expr Expr) |
         (sub Type Expr Expr) |
         (mul Type Expr Expr) |
         (div Type Expr Expr) |
         (mod Type Expr Expr) |
         (shr Type Expr Expr) |
         (shl Type Expr Expr) |
         (bitand Type Expr Expr) |
         (bitor Type Expr Expr) |
         (bitnot Type Expr Expr) |
         (bitxor Type Expr Expr) |
         (eq Expr Expr) |
         (neq Expr Expr) |
         (le Expr Expr) |
         (lt Expr Expr) |
         (cast Type Expr) |
         (conv Type Expr) |
         Call

BranchValue ::= Number | CharLiteral | Symbol
BranchRange ::= BranchValue | (range BranchValue BranchValue)
BranchRanges ::= (ranges BranchRange+)

VarDecl ::= (var SymbolDef VarPragmas Type [Empty | Expr])
ConstDecl ::= (const SymbolDef VarPragmas Type Expr)
EmitStmt ::= (emit Expr+)

Stmt ::= Call |
         VarDecl |
         ConstDecl |
         EmitStmt |
         (asgn Lvalue Expr) |
         (if (elif Expr StmtList)+ (else StmtList)? ) |
         (while Expr StmtList) |
         (case Expr (of BranchRanges StmtList)* (else StmtList)?) |
         (lab SymbolDef) |
         (jmp Symbol) |
         (ret Expr) # return statement

StmtList ::= (stmts Stmt*)

Param ::= (param SymbolDef ParamPragmas Type)
Params ::= Empty | (params Param*)
```

```
ProcDecl ::= (proc SymbolDef Params Type ProcPragmas [Empty | StmtList])

FieldDecl ::= (fld SymbolDef FieldPragmas Type)

UnionDecl ::= (union Empty FieldDecl*)
ObjDecl ::= (object [Empty | Type] FieldDecl*)
EnumFieldDecl ::= (efld SymbolDef Expr)
EnumDecl ::= (enum Type EnumFieldDecl+)

ProcType ::= (proctype Empty Params Type ProcTypePragmas)

IntQualifier ::= (atomic) | (ro)
PtrQualifier ::= (atomic) | (ro) | (restrict)

Type ::= Symbol |
         (i IntBits IntQualifier*) |
         (u IntBits IntQualifier*) |
         (f IntBits IntQualifier*) |
         (c IntBits IntQualifier*) | # character types
         (bool IntQualifier*) |
         (void) |
         (ptr Type PtrQualifier) | # pointer to a single object
         (flexarray Type) |
         (aptr Type PtrQualifier) | # pointer to an array of objects
         ProcType
ArrayDecl ::= (array Type Expr)
TypeDecl ::= (type SymbolDef TypePragmas [ProcType | ObjDecl | UnionDecl |
EnumDecl | ArrayDecl])

CallingConvention ::= (cdecl) | (stdcall) | (safecall) | (syscall)  |
                      (fastcall) | (thiscall) | (noconv) | (member)

Attribute ::= (attr StringLiteral)
ProcPragma ::= (inline) | (noinline) | CallingConvention | (varargs) |
(was Identifier) |
               (selectany) | Attribute
ProcTypePragma ::= CallingConvention | (varargs) | Attribute

ProcTypePragmas ::= Empty | (pragmas ProcTypePragma+)
ProcPragmas ::= Empty | (pragmas ProcPragma+)

CommonPragma ::= (align Number) | (was Identifier) | Attribute
VarPragma ::= CommonPragma | (tls)
VarPragmas ::= Empty | (pragmas VarPragma+)

ParamPragma ::= (was Identifier) | Attribute
ParamPragmas ::= Empty | (pragmas ParamPragma+)

FieldPragma ::= CommonPragma | (bits Number)
FieldPragmas ::= (pragmas FieldPragma+)

TypePragma ::= CommonPragma | (vector Number)
TypePragmas ::= Empty | (pragmas TypePragma+)
```

```
ExternDecl ::= (imp ProcDecl | VarDecl | ConstDecl)

TopLevelConstruct ::= ExternDecl | ProcDecl | VarDecl | ConstDecl |
                      TypeDecl | EmitStmt
Include ::= (incl StringLiteral)
Module ::= (stmts Include* TopLevelConstruct*)
```

Notes:

- `IntBits` is either 8, 16, 32, 64, etc. or the identifier `M` which stands for **m**achine word size.
- There can be more calling conventions than only `cdecl` and `stdcall`.
- `case` is mapped to a `switch` but the generation of `break` is handled automatically.
- `ro` stands for `readonly` and is C's notion of the `const` type qualifier. Not to be confused with NIFC's `const` which introduces a named constant.
- C allows for `typedef` within proc bodies. NIFC does not, a type declaration must always occur at the top level.
- String literals within `emit` produce verbatim C code, not a C string literal.
- For `array` the element type comes before the number of elements. Reason: Consistency with pointer types.
- `proctype` has an Empty node where `proc` has a name so that the parameters are always the 2nd child followed by the return type and calling convention. This makes the node structure more regular and can simplify a type checker.
- `varargs` is modelled as a pragma instead of a fancy special syntax for parameter declarations.
- The type `flexarray` can only be used for a last field in an object declaration.
- The pragma `tls` is used to denote thread local storage. It can only be used on toplevel (aka "global") variables.
- The pragma `selectany` can be used to merge proc bodies that have the same name. It is used for generic procs so that only one generic instances remains in the final executable file.
- `attr "abc"` annotates a symbol with `__attribute__(abc)`.
- `cast` might be mapped to a type prunning operation via a `union` as C's aliasing rules are broken.
- `conv` is a value preserving type conversion between numeric types, `cast` is a bit preserving type cast.
- `array` is mapped to a struct with an array inside so that arrays gain value semantics. Hence arrays can only be used within a `type` environment are become nominal types. A NIFC code generator has to ensure that e.g. `(type :MyArray.T . (array T 4))` is only emitted once.
- `type` can only be used to introduce a name for a nominal type (that is a type which is only compatible to itself) or for a proc type for code compression purposes. Arbitrary aliases for types **cannot** be used! Rationale: Implementation simplicity.

Inheritance

NIFC directly supports inheritance in its object system. However, no runtime checks are implied and RTTI must be implemented manually, if required.

- The `object` declaration allows for inheritance. Its first child is the parent class (or empty).

- The `dot` operation takes a 3rd parameter which is an "inheritance" number. Usually it is `0` but if the field is in the parent's object it is `1` and it is `2` for the parent's parent and so on.

Declaration order

NIFC allows for an arbitrary order of declarations without the need for forward declarations.