NIF data format

The NIF data format is a text based file format designed for compiler frontend/backend communication or communication between different programming languages. The design is heavily tied to Nim's requirements. However, the design works on language agnostic ASTs and is so extensible that other programming languages will work well with it too.

A NIF file corresponds to a "module" in the source language. The module is stored as an AST. The AST consists of "atoms" and "compound nodes".

Design goals

- Simple to parse.
- Simple to generate.
- Text representation that tends to produce small code.
- Extensible and easily backwards compatible with earlier versions.
- Lossless conversion back to the source code is possible.
- Can be as high level or as low level as required because statements, expressions and declarations can be nested arbitrarily.
- Lots of search&replace like operations can be performed reliably with pure text manipulation. No parsing step is required for these.
- Readable and writable for humans. However, it is not optimized for this task.

Extensibility is primarily achieved by using two different namespaces. One namespace is used for "node kinds" and a different one for source level identifiers. This does away with the notion of a fixed set of "keywords". In NIF new "keywords" ("node kinds") can be introduced without breaking any code.

Why yet another data format?

Other, comparable formats (LLVM Bitcode or text format, JVM bytecode, .NET bytecode, wasm) have one or more of the following flaws:

- More complex to parse.
- Too low level representation that loses structured control flow.
- Too low level representation that implies the creation of temporary variables that are not in the original source code file.
- Binary formats that make it harder to create tools for.
- Less flexible.
- Cannot express Nim code well.
- Produces bigger files.

Example NIF module

In order to get a feeling for how a NIF file can look, here is a complete example:

```
(.nif24)
(stmts
(imp 2,5,sysio.nim(type :File (object ..)))
(imp (proc :write.1.sys . (pragmas varargs) (params (param f File)).))
(call write.1.sys "Hello World!\0A")
)
```

A generator can produce shorter code by making use of `.k` and `.i` (substitution) directives:

```
(.nif24)
(.k I imp)
(.k P pragmas)
(.i write write.1.sys)
(stmts
(I 2,5,sysio.nim(type :File (object ..)))
(I (proc :write . (P varargs) (params (param f File)).))
(call write "Hello World!\0A")
)
```

Encoding

A NIF file is stored as a mere sequence of bytes ("octets"). No Unicode validation steps are performed and no BOM-prefix can be used.

Whitespace

Whitespace is used to separate tokens from each other but apart from that carries no meaning and a NIF parser is supposed to ignore whitespace. Editors and other tools can format and layout NIF code to be pleasing to look at.

Whitespace is the set `{' ', '\t', '\n', '\r'}`.

Control characters

NIF uses some characters like `(`, `)` and `~` to describe the AST. As such these characters **must not** occur in string literals, char literals and comments so that a NIF parser can skip to the enclosing `)` without complex logic.

The control characters are:

```
( )  [ ]  { }  ~  #  '  "  \  :
```

Escape sequences

Grammar:

```
HexChar ::= [0-9A-F]
Escape ::= '\' HexChar HexChar
```

String and character literals support escape sequences via backslashes quite like in other languages. Unlike other languages only `\xx` where `xx` stands for the ASCII value that is encoded is supported. Characters of a value < 32 (space) have to be encoded as `\xx` too.

For example, a binary zero in a string literal is written as `"\00"`.

Caution: The commonly used `"\\"` in other languages that escapes the backslash itself is not supported either and must be written as `\5C`!

Rationale: Ease of implementation.

Atoms

Empty

```
Empty ::= '.'
```

As a special syntactic extension, the "empty" or "missing" node is written as a single dot `.`. Empty nodes are frequently used because a construct like `let` can have optional parts like pragmas, a type annotation or an initial value. The empty node is then used to ensure that a `let` node (for example) always has a fixed number of children and that the N-th child is always a type or empty.

Empty nodes do not require whitespace in between them: `...` is a list of 3 empty nodes.

Identifiers

The most common atom is the "identifier". Its spelling must adhere to the grammar:

```
IdentStart ::= <unicode_letter> | '_' | Escape
IdentChar ::= IdentStart | [_0-9]
Identifier ::= IdentStart+ IdentChar*
```

Identifiers have no real meaning, in particular it **cannot** be assumed that two identifiers of the same string (for example `abc`) refer to the same entity.

Identifiers that contain characters that are neither letters nor digits must be escaped via backslashes, `\xx` much like it is used in string and character literals.

Symbols

A "symbol" is a name that refers to an entity unambiguously. A symbol must adhere to the grammar:

```
Symbol ::= IdentStart+ '.' (IdentChar | '.')*
SymbolDef ::= ':' Symbol
```

Roughly speaking that is a "word" that must contain a dot but cannot start with a dot.

For example the 2nd proc named `foo` in a Nim module `m` would typically become `foo.2.m` in NIF.

Symbols that contain characters that are neither letters nor digits must be escaped via backslashes, `\xx` much like it is used in string and character literals.

A `SymbolDef` is a symbol annotated with a leading ':'. It indicates that the parent node is the node introducing this symbol. Thus a tool can implement a feature like "goto definition" in a language agnostic way without having to know which node kinds introduce new symbols.

Numbers

Grammar:

```
Digit ::= [0-9]
FloatingPointPart ::= ('.' Digit+ ('E' '-'? Digit+)? ) | 'E' '-'? Digit+
```

```
Number ::= ('+' | '-') Digit+ (FloatingPointPart | 'u')?
```

Numbers must start with a plus or a minus and only their decimal notation is supported. For example Nim's `0xff` would become `256`.

Unsigned numbers always have a `u` suffix. Floating point numbers must contain a dot or `E`. Every other number is interpreted as a signed integer.

Note that numbers that do not start with a plus nor a minus are interpreted as "line information". See the corresponding section for more details.

Char literals

Grammar:

```
VisibleChar ::= ASCII value >= 32 but not a control character
CharLiteral ::= '\'' (VisibleChar | Escape) '\''
```

Char literals are enclosed in single quotes. The only supported escape sequence is `\xx`.

String literals

Grammar:

```
EscapedData ::= (VisibleChar | Escape | Whitespace)*
StringLiteral ::= '"' EscapedData '"'
```

String literals are enclosed in double quotes. The only supported escape sequence is `\xx`. Whitespace, even including newlines, can be part of the string literal without having to escape it.

For example:

```
  "This is a single\20
  literal string"
```

Produces: `"This is a single \n literal string"`.

Compound nodes

Grammar:

```
Atom ::= Empty | Identifier | Symbol | SymbolDef | Number | CharLiteral |
         StringLiteral

NodeKind ::= Identifier

Node ::= Atom | CompoundNode
NodePrefix ::= LineInfo? Comment?
CompoundNode ::= NodePrefix '(' NodeKind Node* ')'
```

The general syntax for a compound node is `(nodekind child1 child2 child3)`.

That means NIF is a Lisp with some extensions:

- The ability to annotate (line, column, filename) information for a node.
- The ability to annotate a node with a comment. (In Lisp comments are not attached to a node.)

Unlike in Lisp a function `call` is not implied so what is usually just `(f a b c)` in Lisp becomes `(call f a b c)` in NIF. The first item in a list (`call` in the example) is called the "node kind". There are many different node kinds and the set of node kinds is extensible.

However, usually at least the following node kinds exist and ensure a minimum of compatibility between programming languages.

Node kind	Description
`nil`	A nil/null pointer. Note: This is not an atom so that it does not conflict with an identifier named `nil` and so that it can get a type annotation.
`false`	The boolean value `false`. Note: This is not an atom so that it does not conflict with an identifier named `false`.
`true`	The boolean value `true`. Note: This is not an atom so that it does not conflict with an identifier named `true`.
`stmts`	A list of statements.
`expr`	A list of statements but ending in an expression.
`imp`	An import of a declaration from a different module.
`proc`	A proc declaration. Note: For Nim `func`, `iterator` etc. are also used.
`type`	A type declaration.
`params`	Wraps a list of parameters.
`param`	A parameter declaration.

Node kind	Description
`var`	A var declaration.
`let`	A let declaration.
`fld`	An object field declaration.
`const`	A const declaration.
`if`	An `if` statement.
`elif`	An `elif` section inside an `if` statement.
`else`	An `else` sectin within an `if` statement.
`while`	A `while` loop.
`ret`	A return statement.
`brk`	A break statement.
`and`	Logical `and` operator.
`or`	Logical `or` operator.
`not`	Logical `not` operator.
`addr`	Address-of operator.
`deref`	Pointer dereference operation.
`asgn`	Assignment statement.
`at`	Array index operation.
`dot`	Object field selection.
`add`	Arithmetic add instruction. Usually the first child is a type like `i32` to specify an `i32` addition.
`sub`	Arithmetic sub instruction. Usually the first child is a type like `i32` to specify an `i32` subtraction.
`mul`	Multiplication. Takes a type like `add`.
`div`	Division. Takes a type like `add`.
`mod`	Modulo operator. Takes a type like `add`.
`shr`	Bit shift to the right. Takes a type like `add`.
`shl`	Bit shift to the left. Takes a type like `add`.
`bitand`	Bitwise and. Takes a type like `add`.
`bitor`	Bitwise or. Takes a type like `add`.
`bitnot`	Bitwise not. Takes a type like `add`.

Node kind	Description
`eq`	Testing for equality. Takes a type like `add`.
`neq`	Testing for "not equals" ("!="). Takes a type like `add`.
`le`	Less than or equals ("<="). Takes a type like `add`.
`lt`	Strictly less than ("<"). Takes a type like `add`.
`i` N	Where N can be 8, 16, ... The signed integer type that uses N bits.
`u` N	Where N can be 8, 16, ... The unsigned integer type that uses N bits.
`f` N	Where N can be 8, 16, ... The floating point type that uses N bits.
`array`	Type constructor that produces an `array` type.
`object`	Type constructor that produces an `object` type.
`ptr`	Type constructor that produces a pointer type.
`proctype`	Type constructor that produces a proc type.
`pragmas`	List of pragmas.
`kv`	A single (key, value) pair. The `ExprColonExpr` node kind in Nim.
`vv`	A (value, value) pair. The `ExprEqExpr` node kind in Nim.
`par`	Wraps an expression inside parentheses.
`cons`	An object/array/etc. constructor. First child is a type.
`lab`	A label declaration (target of a `jmp`).
`jmp`	A jump or goto instruction.

Line information

Grammar:

```
LineDiff ::= Digit* | '~' Digit+
LineInfo ::= LineDiff (',' LineDiff (',' EscapedData)?)?
```

Every node can be prefixed with a digit or `~` or `,` to add source code information. ("This node originates from file.nim(line,col).") There are 3 forms:

1. `<column-diff>`
2. `<column-diff, line-diff>`
3. `<column, line, filename>`

The `diff` means that the value is relative to the parent node. For example `8` means that the node is at the same position as the parent node except that its column is `+8` characters. Negative numbers use the tilde and

not the minus. Negative numbers are usually required for "infix" nodes where the left hand operand preceeds the parent (`x + y` becomes `(infix add ~3 x 2 y)` because `x` is written before the `+` operator).

The AST root node can only be annotated with the form `<column, line, filename>` as it has no parent node that column and line could refer to.

Note that numeric literals in NIF have to start with `+` or `-` and thus cannot cause ambiguity with line information.

Since the information includes both lines and columns it can easily take up 10-20% of the file size. Therefore a mere digit starts a line information and not a numeric literal. Numeric literals are not nearly as frequent in practice.

Comments

Grammar:

```
Comment ::= '#' EscapedData '#'
```

Every node can be prefixed with `#` to add a comment to the particular node. The comment also has to end with a `#`.

For example:

```
# This performs an add.#(add x y)
```

Note how the comment ends at `#`. This is not ambiguous as any control character within a comment would have to be escaped via `\xx`.

If a node is annotated both with line information and a comment the line information has to come first.

Directives

A directive looks like `(.directive ...)`. This is not ambiguous because a node kind cannot start with a dot. The existing directives are:

- `.nif<version>`
- `.vendor`
- `.platform`
- `.config`
- `.dialect`
- `.i` and `.k` substitutions.

Directives must be at the start of the file, before the module's AST. Directives that unknown to a parser should be ignored. An implementation can offer additional directives.

Rationale: Compatibility between different NIF versions and implementations.

Version directive

The version directive looks like `(.nif<version>)`. Version is currently always `24` because the first version of this NIF spec was released in 2024.

For example:

```
(.nif24)
```

There must be no whitespace before the version directive so that it also functions as a "magic cookie" for tools that use these to determine file types.

Dialect directive

The `.dialect` directive takes a single string literal describing what exactly the NIF code describes. The values are vendor specific. For Nim the possible values are:

Value	Description
`"nim-parsed"`	Parsed Nim code without semantic checking.
`"nim-gear2"`	Parsed Nim code after semantic checking.
`"nim-gear3"`	Nim code after inlining.
`"nim-gear4"`	Nim code after destructor injections (final step before code generation).
`"nif-c"`	NIF code that is very close to C code.

Substitutions

Every token in the class `{identifier, symbol, node-kind}` can be subject to a simple token substitution mechanism. For identifiers and symbols a substitution looks like `(.i <name> <to be replaced with>)`.

For node kinds a substitution looks like `(.k <name> <node kind>)`.

The tree that is a result of a substitution **must not** itself contain a substitution. This also implies that `(.i a a)` does not cause an infinite regress: The identifier `a` is simply replaced by `a` and the substitution stops afterwards.

Rationale: This allows for a user to come up with a scheme like "every substitution name ends in a digit and if an identifier already ends in one we generate (.i a0 a0)".

For example:

```
(.i ECHO echo.1.system)
(.k C call)
(.i Hello "Hello world!\0A")

(stmts
(call ECHO +1 +2 +3)
```

```
  (C ECHO Hello)
  )
```

Other directives

These look like `(.vendor "some string here")` and `(.platform "some string here")` and `(.config "some string here")` and contain vendor specific information. Usually these can be ignored.

Modules

A complete NIF module consists of a list of directives followed by other CompoundNodes. Typically, there is a single root node of kind `stmts`.

Formally a module is simply a non-empty list of `CompoundNode`:

```
NifModule ::= CompoundNode+
```

NIF trees as identifiers

In many cases it is useful to turn a NIF tree into a canonical string representation that also forms a valid identifier (for C code generation or otherwise). The following encoding scheme accomplishes this task:

1. Line information and comments are ignored.
2. The unary `+` for numbers are removed.
3. The substring of trailing `)` is removed as there is nothing interesting about `))))`.
4. Whitespace is canonicalized to a single space.
5. The space after `)` and before `(` is removed.

`(` is turned into `A`. `)` is turned into `Z`. A space that separates the children of a compound node is turned into `S`.

The empty node (`.`) is encoded as `E`.

Note that `.` within a symbol is **not** escaped!

The N-th (where N > 1) occurance of a symbol or identifier is encoded as `R<x>` where `x` refers to the `x`'s symbol or identifier that already been emitted. But only if `R<x>` is still shorter than the symbol/identifier.

For example:

`(tag abcdef . abcdef)` is encoded as `AtagSabcdefSESR0`.

Characters like `A` and `Z` that are used in the encoding must be escaped should they occur. The encoding is `X<xx>` where `xx` is the hexadecimal value of the character's byte value. Other characters that are not valid identifiers such as the space character or a newline are encoded as `X<xx>` too.

In summary:

Letter	Used for

Letter	Used for
A	begin of a compound node (
Z	end of a compound node)
E	the empty node
S	space; separator between a node's children
O	encodes the colon in a SymbolDef
U	encodes the " that is used to delimit string literals
X	used to escape the letters used in this encoding and in general for characters that should not be used in an identifier
R	reference to an identifier or a symbol that has already occured
K	reference to a node kind that has already occured

For example:

```
(array (range +0 +9) (array (range +0 +4) (i +8))))
```

Becomes:

```
(array(range 0 9)(array(range 0 4)(i 8
```

Which then is turned into:

```
AarrayArangeS0S9ZAK0AK1S0S4ZAiS8
```