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

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.

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 ':' that indicates this 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 model the AST precisely.

Numbers

Grammar:

```
Digit ::= [0-9]
NumberSuffix ::= [a-z]+ [0-9a-z]*  # suffixes can only contain lowercase letters
FloatingPointPart ::= ('.' Digit+ ('E' '-'? Digit+)? ) | 'E' '-'? Digit+
Number ::= '-'? Digit+ FloatingPointPart? NumberSuffix?
```

Numbers must start with a digit (or a minus) and only their decimal notation is supported. Numbers can have a suffix that has to start with a lowercase letter. For example Nim's `0xff'i32` would become

`256i32x`. (The `x` encodes the fact that the number was originally written in hex.)

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:

```
Newline ::= Whitespace* '\n' Whitespace*
StringSuffix ::= Identifier
EscapedData ::= (VisibleChar | Escape | Newline)+
StringLiteral ::= '"' EscapedData '"' StringSuffix?
```

String literals are enclosed in double quotes. The only supported escape sequence is `\xx`. A string literal can contain newlines for better readability. Such newlines including any preceeding and succeeding whitespace is ignored. Note that a literal newline character would be written as `\0A` so that this is not ambiguous.

For example:

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

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

A string literal can have a suffix that is usually ignored but can be used to store the original format of the string. For example, Nim supports "raw string literals" and "triple string literals". These could be modelled as `R` and `T` suffixes:

```
  "This was a triple quoted Nim string"T
```

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`.
`ge`	Greater than or equals (">="). Takes a type like `add`.
`gt`	Strictly greater 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.
`pragma`	A single (key, value) pragma pair.
`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 `@` 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 are valid too and 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.

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`
- `.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)
```

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 a single AST that starts with a `stmts` root node.

But formally it is simply a non-empty list of `CompoundNode`:

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