NML

Language Description (draft)

v0.4

-

-

29 Sep 2015

`http://www.generaldevelopment.com.au`

Prepared by Daniel Kos, General Development Systems, October 2015.

Typeset using LaTeX.

Contents

Chapter 1

Introduction

1.1 Version History

- In v0.4 of this document, we have removed support for mixed-case hexadecimal literals. The previous arrangement allowed for hexadecimal digits >9 to be entered in lowercase, which allowed for hex literals that looked like identifiers, affecting code readability.

- In v0.3 of this document, we have changed CData syntax from being encapsulated by a single brackets [ ] to brackets and braces [{ and }]. The previous arrangement meant that a string of CData could never end with a backslash.

1.2 About NML

NML stands for New Multipurpose Language. The initials are also something of a loose pun on XML – the language from which NML draws its inspiration.

Like XML, NML is not a complete language in itself but rather a gereralised 'base' language, defining a core syntax on which to build complete languages (in the same way as HTML is built on XML syntax). NML does not define any specific language behaviour or implementation.

1.3 NML vs XML

NML is based loosely on XML (XML code can be converted to NML very easily, see appendix A), but adds an extra dimension of data nesting, as well as non-string data types and value lists. While XML's applications are limited to document markup and hierarchical data organisation, NML is intended to be suitable for a broader set of applications (as we aim to demonstrate with examples in this document), including functional programming.

1.3.1 What's wrong with XML?

XML is excellent for what it was designed for: pure markup applications, with HTML being the most ubiquitous, but it can't really be extended to other programming paradigms. (As an example, consider web development. A modern web developer has to learn not one but three languages: HTML, CSS, and JavaScript. No one set of syntax rules fulfills all three roles on its own.)

In our case, we wanted to develop a functional programming language with XML syntax. After trying in vain to devise a workable syntax for what we wanted to achieve, that would also meet W3C's XML specification, we eventually gave up on the idea and developed NML instead.

1.3.2 What about JSON?

JSON (JavaScript Object Notation) already provides much the same data-organisation flexibility as NML, but unfortunately is rather unwieldly when it comes to applications outside data organisation. It wouldn't be particularly user-friendly as a document markup language, for instance.

NML falls somewhere between XML and JSON. You can think of NML as a sort of compromise between the expressive abilities of the two languages.

1.4 Example syntax

These examples introduce the reader to various language elements, while at the same time demonstrating the expressiveness of NML. All examples are valid NML code. However, the tag and field names, as well as implied behaviours of the following examples, are hypothetical only. NML provides the syntax framework only. It does not specify tag and field names, nor runtime behaviour.

1.4.1 Example 0: Hello world

```
Hello, world!
```

Text entry is the 'root' entry mode of NML. Unlike XML, there is no need for a root tag.

(There is however a 'root field'. Fields will be introduced in the next example.)

1.4.2 Example 1: encoding web-like page markup

```
<web title={Welcome} description={A sample page} body={
    <heading level=1 {Sample page}>
    This is a sample of what an NML-based alternative
    to HTML might look like.
    <\p><hyper url={http://www.example.com/} {Click here!}>
}>
```

This purely hypothetical example demonstrates how web pages might be encoded if they were based on NML rather than XML syntax.

Lets identify the various language elements in this example:

- `web`, `heading`, and `hyper` are *identifiers*, or names, for different kinds of *tags*. Tags mark-up and give structure to data.
  - Tags are encapsulated by angle brackets < >.

- `title`, `description`, `body`, `level`, and `url` are identifiers for different *fields* within tags. Fields store data.
- Items that follow an = sign are *field elements*. Field elements are elements of data within fields.
- `{Sample page}` and `{Click here!}` are also field elements, even though they do not have an associated field identifier or = sign. Each tag is allowed to have a single *default field*, which is unnamed. Field elements are automatically assigned to the default field if no field identifier is used.
- Braces `{ }` encapsulate text data, which can contain embedded tags, among other things.
- `<\p>` is not a tag but an escape sequence. Escape sequences allow entry of special characters. This particular escape sequence inserts a character that begins a new paragraph.

1.4.3 Example 3: procedural programming

```
<function main {
    <set name, <input {Enter your name...}>>
    <print {Hello, }, name, {!}>   <!e.g. 'Hello, Jim!' !>

    <print {Let's count to 10!}>
    <for i,1,10 do={           <!loop over values 1 to 10!>
        <print i>
    }>

    <return 0>
}>
```

This example demonstrates how a procedural language might look if expressed in terms of NML syntax. The syntax is rather more compact and workable than it might be had we attempted to express it in XML or JSON syntax.

This example makes heavy use of default fields, and we also introduce the reader to NML's ability for a field to store multiple field elements, represented by a comma separated list.

- `main`, `name`, and `i` are examples of identifiers that are neither tag names nor field names. Rather, they are used here as field element *values* within the default fields of various tags.
- Pieces of text enclosed between `<!` and `!>` are *comments*. They explain the code to the reader without affecting functionality.

1.4.4 Example 4: Structured data representation

```
<customer
    name={Smith},{John}
    address=123,{Sample St},null,{Townsville},{QLD}
    dob=<date y=1942 m=6 d=24>
>
```

Although XML is a popular choice for structured data representation, NML provides a bit of additional flexibility. In XML, each *attribute* is limited to storing a single string. NML allows for each *field* to store any number of ordered data elements, as well as nested tags.

Chapter 2

Data storage and oragnisation

This chapter covers how data are sored and organised internally.

2.1 Data types

Any field element may store one of the following data types.

2.1.1 String

A string is a sequence of characters, used to store such things as text, phone numbers, and occasionally non-readable data.

2.1.2 Integer

An integer is a whole number, which does not have a fractional part. How integers are stored, and the range of valid integers, is machine and/or implementation dependent. Integers may be positive or negative (there are no ‘unsigned’ integers).

2.1.3 Floating point number

A floating point number is a number with a fractional part (for example 3.14159). Although most system architectures support two different floating point precisions, NML syntax doesn’t allow any distinction. Implementors should use double precision where practical. (Nominally, the minimal byte-size of a tagged union implementation of field elements will negate any space saving from using single precision floats anyway.)

It is worth noting that floating point storage provided by most system architectures is based on binary representation. Many decimal fractions do not have a rational binary representation, and are subject to rounding error when stored in this format. In particular, binary floating point representation is inadequate for storing currency values. Implementors may opt to implement floating point numbers with an internal decimal representation to mitigate this issue, although this incurs a performance overhead.

2.1.4 Identifier

An identifier is a character sequence, such as a keyword, that is used to identify something within the language, such as the name of a tag, field, or something else.

Identifiers are not strings: although externally they are entered as a sequence of characters, a real-world implementation is allowed to convert them internally to a more efficient internal representation from which the original character sequence cannot be recovered.

2.1.4.1 Reserved identifiers corresponding to literal values

The following identifiers are reserved within NML, and correspond to non-numeric, non-string, literal values. This is a minimal set. Implementors are free to expand this list to allow for additional literal values, such as 'not_applicable' for survey data, for instance.

Identifier	Definition
`null`	Empty field element. Default for undefined fields.
`inf`	Result of division by zero. (Conceptually infinite.)
`invalid`	The result of an invalid operation (e.g. multiplication by a string).
`true`	States of Boolean truth and falsehood.
`false`	

2.1.5 Tag

Tags are data structures containing one or more fields.

2.1.5.1 Fields

A field stores a list of one or more values, each of which is known as a 'field element.'

Different field elements within a field can store different data types.

All fields are capable of storing multiple elements even though many fields will only require a single value.

Each field element can contain a tag, an identifier, or literal data (including states like `null` and `invalid`).

Analogous to...

- A single variable, or an array of variables, in a procedural language.
- A table column in a relational DBMS.
- Attributes in XML.
- Tag content in XML.

Special fields

- Tag type field: one or more field elements that define the tag type.
- Default field: optional field with null identifier (unnamed).

2.1.5.2 Tags

A tag is a collection of named fields, as well as an optional unnamed default field, and a tag type (which is also a field).

Analogous to...

- a struct in C language.
- A record in BASIC.
- a table row in a relational DBMS.

2.1.5.3 Identifiers

Identifiers are used to specify field names and tag types. An identifier is composed of any sequence of characters (excepting the null character). Identifiers are independent of NML syntax rules.

Even though identifiers are made of characters, they are not strings and cannot be evaluated as strings. They are merely mnemonic tokens for the parser to recognise and convert into some internal representation.

Identifiers represent tag types and field names. They can also be assigned as values to field elements (however, this specification does not dictate how an identifier used as a value should be interpreted).

Validity of identifiers:
Any sequence of characters (excepting the null character) is a valid identifier. Escape sequences may be used to input characters that would otherwise conflict with language syntax (such as spaces, or the tag opening bracket '<', or identifiers that resemble literal data), or to allow people in English-speaking countries to enter identifiers that contain non-western symbols, for example.

Rationale:
it is our view that particular identifiers should not be forbidden by or dependent on any particular language syntax (especially as real-world NML implementations may impose their own additional syntax rules). Even if an identifier may be inconvenient to type with escape sequences, it should still be possible to enter it into the system if needed. (In the same way, UNIX filenames aren't restricted by syntax rules of the various command line shells available.) Nor should identifier naming rules discriminate against users of non-english alphabets.

Chapter 3

Syntax

3.1 Text entry

Text entry is the default entry mode in NML. This means that if your code consists of pure text at the top level, such as `Hello, world!`, without an unescaped `<` character, then this text will be assigned to the first field element of the root field of the NML input. Below the top level, you can enter text anywhere a field element can be specified, by enclosing it in braces `{ }`.

You can embed tags, escape sequences, and comments inside text. Because a single field element cannot store both a text string and a tag, text with embedded tags is stored in multiple field elements. This means that:

```
<tag field={Hello, <getname>!}>
```

is equivalent to:

```
<tag field={Hello, }, <getname>, {!}>
```

You can separate a string into two separate elements without a closing brace using the string element separator `<>`.

3.1.1 Whitespace handling

Whitespace consists of spaces, tabs, and newlines. Whitespace is useful for organising and indenting code for clarity, although one doesn't want these extra spaces, newlines, and tabs to form part of the data parsed into the system.

We therefore specify that NML collapses whitespace (the exception being inside CData). Any amount of contiguous whitespace is reduced to a single space character. This behaviour is similar to that adopted by HTML. Escape sequences are available for making newlines, tabs, and spaces explicit.

3.2 Tags

These are loosely based on XML tags. You open a tag with a `<` and close it with a `>` . Unlike XML, tags are self-contained. There is no such thing as an end-tag in NML.

Tags contain a type field, and zero or more additional fields, including an optional default field, which has no field identifier.

```
<type field1=value field2=value1, value2 value ....>
```

The last value above is attached to the default (unnamed) field. Any field can have multiple field elements, separated by commas , . Default field values do not need to be separated by commas (although it certainly doesn't hurt to do so), since the lack of a comma after any value means we revert to the default field, until we read another field name (which we recognise as an identifier followed by an = sign).

Optional whitespace may be used to pad out commas, equals signs, and angle brackets.

3.3 Tag type field

The type field is the first thing specified after opening a tag. There is no field identifier or = sign; type field elements always follow a tag opening.

As with other fields, multiple elements can be specified, separated by commas. This allows namespace hierarchies, special modifier symbols (specified as identifiers), among other things. The type field behaves just like any other field.

The first example tag has two elements in its type field (an example of a namespace hierarchy). The second and third examples have their type fields defined by a single element.

```
<gui,button {Hello world!} name={blah}>
<set customers,7254,address val={221B Baker St}>
<action on=blah,click <quit>>
```

3.4 Default field

Input reverts to the default field if a data element is specified that is not preceded by the beginning of a tag < , a comma , or an equals sign = . Any data types can be used, including identifiers. An identifier that is not followed by an equals sign is treated as a value of a field element rather than a field identifier. Multiple default field elements may be separated by commas, but they don't have to be, since the lack of a comma causes the parser to revert to appending value elements to the default field anyway.

3.5 CData

CData, or Character data, provides a way to escape an entire string of text. A string of CData begins with [{ and is terminated with }] .

Unlike regular text entry, all whitespace is preserved, symbols like < and > are *ignored*, comments are *left in*, etc.

The caveat is that within cdata, the symbol] must be escaped as \] . This ensures that any freak occurance of }] in the middle of your data is not mistaken for the end of the CData. A backslash that isn't followed by] is given no special treatment, meaning that if you want \] to appear in you data without being reduced down to] , you can specify it as \\] , and the first backslash will remain intact.

The best approach is to sanitise CData through a filter before sending it to an NML implementation. This is simply a matter of scanning for each instance of `]` and replacing it with `\]` (there is no need to even look at the surrounding characters).

3.5.1 Rationale for requiring both brackets and braces

The alternative, enclosing CData between square brackets alone, would mean that CData could never end with a backslash.

3.6 Comments

Comments allow the insertion of arbitrary text into NML code without it having any effect on code or data. They otherwise perform no function; the parser simply skips over them as though they weren't there.

3.6.1 Purpose

Comments are typically short explanatory notes left for the benefit of code maintainers. They also provide a way of disabling code without removing it entirely.

3.6.2 Syntax

- `<!`*comment text*`!>`

Token	Definition
`<!`	Denotes start of comment
`!>`	Terminates comment
comment text	Any character sequence that doesn't contain `!>`

When the parser encounters a `<!` sequence it will ignore that sequence and everything that follows until it reads a `!>` sequence.

Comments are valid everywhere except in cdata (where they are treated as literal data) and inside other comments (comments do not nest). They are valid within strings. They are even valid in the middle of identifiers (mainly for reasons relating to consistency with escape sequences, although placing a comment in the middle of an identifier is always an unusual thing to do).

3.6.2.1 Example

```
<!this is a comment that will be <completely> {ignored}
                                     by the parser!>
```

3.7 Escape sequences

An escape sequence is a means of entering a particular character while overriding its syntactical meaning within the language (for example, we might wish to use a right brace `}` in the middle of a text element, without the parser interpreting it as the end of the text). Another use for escape sequences is for entering characters that aren't present on the user's keyboard.

3.7.1 Syntax

On the surface, escape sequences resemble tags, but with a terser form. (Unlike tags, escape sequences represent a single character rather than a field element.

You can use an escape sequence just about anywhere, including in an identifier.

Token	Meaning
<\	begin escape sequence
>	terminate escape sequence

Escape sequences cannot contain spaces. The sole exception is the <\ > sequence, which escapes a space character.

3.7.1.1 Escaping common symbols

Any character may be escaped simply by placing it between <\ and > without spaces, with the exception of characters `a–z` and `#` , which are reserved for special escape sequences and cannot be escaped themselves.

Thus <\<> produces the escaped form of < , which will be treated like a normal character, rather than having a special meaning within NML. Similarly, <\ > escapes a space, allowing the use of spaces within identifiers.

3.7.1.2 Entering a character by unicode

- <\#*charcode*>

charcode can be any unicode character code. It is expressed using NML's numerical literal syntax (described later), which means it can be expressed in whichever number system you prefer (e.g. octal or hexadecimal, with decimal being the default).

3.7.1.3 Other escape sequences

Characters a-z and A-Z are reserved for special escape sequences, most of which have not been defined yet.

Escape sequence	Definition
`<\n>`	newline (line break)
`<\p>`	new paragraph (paragraph break)
`<\t>`	tab character
`<\s>`	zero-width space
`<\S>`	non-breaking space
`<\h>`	Soft hyphen
`<\H>`	non-breaking hyphen
`<\d>`	en dash
`<\D>`	em dash

3.7.2 Why are escape sequences required?

- To make the parser overlook characters that would otherwise have a special meaning in NML, and treat them just like any other characters. You might want to have a space in the middle of an identifier (this kind of naming

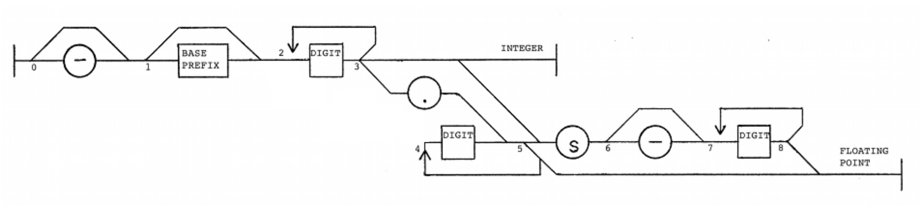

Figure 3.1: Syntax diagram for numerical input. If a token breaks the numerical input syntax pattern, it is thus determined to be an identifier rather than a number literal.

is not recommended) or you might want to insert a less-than symbol, < , into a string, without NML thinking you mean to begin a tag.

  - to make special characters like '<' lose their power, so you can use them in identifiers and strings just like any other character.
  - To make NML treat numbers like characters, rather than digits.

- To make things explicit, like spaces and newlines, which NML might otherwise treat as spurious whitespace in code layout rather than intentional formatting

- To occasionally enter characters you don't have on your keyboard, and invoke identifiers that contain such symbols

3.8 Numerical input syntax

NML accepts numerical literals entered in binary, octal, decimal (default), and hexadecimal, with optional scientific notation.

3.8.1 Parts of input

A numerical literal consists of the following components, in order, without spaces.

	Part		Example
1	Negative sign	*(optional)*	`-`
2	Base prefix	*(optional)*	`x`
3	Mantissa	*(mandatory)*	`27.2421`
4	Exponent	*(optional)*	`s-3`

3.8.1.1 Negative sign

If the number is negative, the first character must be a negative '-' sign.

If there is more than one negative sign, an even number denotes a positive number and an odd number denotes negative.

Negative signs must precede any base prefix.

3.8.1.2 Base prefix

If a number is not in decimal form then a base prefix should be specified after any negative sign and before the mantissa. The prefix must be lowercase.

Prefix	Base	Number system
`b`	2	Binary
`o`	8	Octal
	10	Decimal (default)
`x`	16	Hexadecimal

3.8.1.3 Mantissa

The main part of the number, which may contain a '.' point character to separate the integer part from the fractional part of the number. If a point is present then the number is represented internally as a floating point number even if there is no fractional part after the point. If the integer part is zero then a '0' must be placed before the point. The mantissa cannot begin with a point character.

Number system	Valid numerals
binary	`0, 1`
octal	`0–7`
decimal	`0–9`
hexadecimal	`0–9, A–F`

Examples:

- `3`
  - This is an integer
- `3.14`
  - This is a floating point number
- `3.`
  - This is a floating point number

3.8.1.4 Place shift (scientific exponent)

A lowercase `s` after the mantissa indicates that an exponent follows.

The exponent is a decimal integer (regardless of mantissa base), which indicates how many places the point should be shifted. An exponent can be prefixed by a negative sign, which indicates a left shift of the point. A positive exponent has no sign, and indicates a right shift.

For binary numbers this is equivalent to a bit-shift operation.

The resulting number, n, is given by

$$n = m \times b^e$$

where m is the mantissa (with any negative sign applied to it), b is the base, and e is the exponent.

3.8.2 Examples

- `-b101s-5` – this is binary $-101_b \times 2^{-5} = -0.00101$.
  - The `s-5` represents a scientific exponent which shifts the point 5 binary places to the left.
- `xFF` – hexadecimal representation of 255.
- `o1224` – octal integer.
- `42` – a humble decimal number can be represented just like you've always done
- `6.02s23` – this means 6.02×10^{23}

3.8.3 Notes

- Programming languages and spreadsheets traditionally use the letter `e` to denote scientific exponent. NML uses the letter `s`, which stands for 'shift.'
  - NML treats different number systems on an equal footing with decimal, meaning that scientific notation can be applied to hexadecimal numbers. In this context, the letter `e` representing exponent would be ambiguous, as `e` is a hexadecimal numeral.
  - The exponent is specified in decimal, regardless of the base used. The exponent represents a power of the base of the number system used (for example, a power of 10 for decimal, or a power of 8 for octal).
- If you use scientific notation, or supply a point (`.`), or specify a number with too many digits to fit in your computer's integer representation, NML will store it as a floating point number. Otherwise, it will be stored as an integer.
  - The base in which numbers are entered has no effect on how they are stored internally.
  - Numerical values are stored as either binary integer or binary double-precision floating point. The number of bits used depends on the system architecture of the user's machine.
- Hexadecimal digits A-F must be entered in uppercase. This requirement was added to address a code readability issue. For example, the hexadecimal numeral `xACED` would look more confusingly like an identifier name if it were allowed to be written as `xaced`.

3.9 Literal identifiers

The following reserved identifiers have a literal meaning. They cannot be used as tag or field identifiers as they always evaluate to literal values.

```
true      false      null      inf      invalid
```

Chapter 4

Additional requirements

4.1 Character encoding

4.1.1 Requirement A

NML code shall be encoded in UTF8.

4.1.1.1 Rationale

- UTF8 is a sensible and space-efficient encoding scheme that is widely used in internet applications.
  - UTF8 uses one byte per character for Unicode character codes 0-127, and two or more bytes per character to encode Unicode characters above 127.
- Plain old standard ASCII (not extended ASCII) is a subset of UTF8, meaning that (in English-speaking countries, at least) most plain text files in existence are already UTF8.
- Code streams dynamically generated by other applications are typically UTF8 already without the programmer having to do any extra work.

4.1.1.2 Exception

If use of UTF8 encoding creates an incompatibility with an established system, another character encoding scheme may be substituted instead. This deviation from the specification shall be well documented. The implementation shall not require, nor accept, any character encoding information in the file header, unless such information is mandated by the character encoding standard used.

4.1.2 Requirement B

No byte-order-mark for UTF8 shall be required (or generated) in a UTF8 encoded NML code file.

4.1.2.1 Rationale

Byte order marks are not recommended by the UTF8 standard as they break backward compatibility, they are not universally recognised or implemented, and are not even particularly useful.

Appendix A

Converting XML data to NML

1. Replace trailing `>` in start tags with `{`
2. Replace end tags with `}>`
3. In self-closing tags, replace `/>` with `>`
4. Replace quotation marks with braces `{ }`
5. Convert special character entities to equivalent NML escape sequences
6. For comments...
   (a) replace `<!--` with `<!`
   (b) replace `-->` with `!>`