Translator NodeBrain Module

Release 0.9.03

Translator NodeBrain Module
December 2014
NodeBrain Open Source Project

Release 0.9.03

Author: Ed Trettevik

History

2005-10-12	Title: *Translator NodeBrain Module* Author: Ed Trettevik <eat@nodebrain.org> Publisher: NodeBrain Open Soure Project
2010-12-31	Release 0.8.3 • Updates - still needed

Preface

This module is intended for readers seeking an introduction to NodeBrain through a series of simple examples. Other documents are available for readers looking for a more complete reference to the rule language, modules, or API (application programmatic interface).

The intent of the examples is to illustrate individual concepts, not to provide complete working applications or show all related options. We avoid formal syntax descriptions, thinking you are here because you want to figure it out from examples.

Files referenced in this tutorial are included in the tutorial directory of the NodeBrain distribution.

See www.nodebrain.org for more information and the latest update to this document.

Documents

NodeBrain Guide - Information on using `nb`
NodeBrain Tutorial - A gentle introduction to `nb` and the rule language
NodeBrain Language - Rule language syntax and semantics
NodeBrain Library - C API

Document Conventions

Sample code and input/output examples are displayed in a monospace font, indented in HTML and Info, and enclosed in a box in PDF or printed copies. Bold text is used to bring the reader's attention to specific portions of an example. In the following example, the first and last line are associated with the host shell and the lines in between are input or output unique to NodeBrain. The `define` command is highlighted, indicating it is the focus of the example. Lines ending with a backslash \ indicate when a command is continued on the next displayed line. This is supported by the language within source files, but not for other methods of command input. If you copy an example of a command displayed over multiple lines, you must enter it as a single line when used outside the context of a source file.

```
$ nb
> define myFirstRule on(a=1 and b=2) mood="happy";
> assert mood="sad";
> show mood
mood = "sad"
> assert a=1,b=2,c=3,d="This is an example of a long single line that",\
    e="we depict on multiple lines to fit on the documnet page";
2008/06/05 12:09:08 NB000I Rule myFirstRule fired(mood="happy")
> show mood
mood = "happy"
> quit
$
```

Table of Contents

1 Concepts

A translator node accepts character strings and, based on regular expression matching, translates them into classification values and/or NodeBrain commands.

2 Tutorial

I don't mind what language an opera is sung in so long as it is a language I don't understand. —Sir Edward Appleton (1892–1965)

NodeBrain supports rules used to translate an opera into NodeBrain commands. Well, okay, not exactly an opera, but lines of text conforming to some foreign syntax for which the significant elements can be recognized and extracted with regular expressions. The desired translation is specified in a NodeBrain translation rule file called a "translator." This capability is made available to node modules via the API. The Translator node module is a simple example of one that uses NodeBrain's translator feature.

2.1 Foreign Text

Let's start this tutorial with a file named `opera.txt` containing the text of a Mother Goose rhyme.

```
# File: tutorial/Translator/opera.txt
#
# Mother Goose Nursery Rhyme
# The Man Who Had Naught
#
There was a man and he had naught,
  And robbers came to rob him;
He crept up to the chimney pot,
  And then they thought they had him.

But he got down on t'other side,
  And then they could not find him;
He ran fourteen miles in fifteen days,
  And never looked behind him.
```

2.2 Translator File

A translator recognizes elements of foreign text and converts it into NodeBrain commands. Let's create one called `opera.nbx` that can be used to translate files that look like the `opera.txt` file.

```
# File: tutorial/Translator/opera.nbx
#

# Ignore lines starting with "#"
(^#)

# Pick up lines with "had" followed by a word,
# but continue on looking for other matches
# even when a match is found.
@(had (\w*)):alert type="Had",what="$[1]";

# Look for "he had" lines - will be subset of "had" lines
(he had (\w*)):alert type="HeHad",what="$[1]";

# Look for lines starting with "And".
(^\s*And ){
  (^then they ){
    (thought)
    :assert info="$[=]";
    }
  (^(\w*) )[$[1]]{
    "robbers":assert info="bad guys $[>]";
    "never":assert info="they didn't $[>]";
    }
  }

# Look for "He <verb>" lines.
(^He (crept|ran) ){
  @"fourteen miles in fifteen days,":assert info="slow runner this man who had naught";
  :assert heVerb="$[1]";
  }
```

The lines starting with an open parenthesis `(` specify a regular expression up to the balanced close parenthesis `)` to be matched against lines of foreign text. The first matching expression determines the translation; that is, the translator stops on a match and subsequent expressions are not evaluated. However, if you start the line with "`@(`" instead, the translator will continue even after a match.

If nothing follows the expression, as with `(^#)` and `(thought)` above, the translator takes no action on a match. This means all lines starting with `#` will translate into nothing. You could say they are ignored or suppressed.

An expression can also be followed by a single action, like the `@(had (\w*))` and `(he had (\w*))` expressions which are followed by "`:alert ...`".

When an expression is followed by "`{`", a nested translator is specified until the following "`}`". A nested translator matches against the text following the previous match. Let's consider the input line "And then they thought they had him." When this line matches `(^s*And )` in the translator, the nested translation block will work on "then they thought they had him." After a match on `(^then they )`, nested translation continues trying to match "thought they had him."

NodeBrain commands are passed to the interpreter with an expression starting with colon (`:`) to distinguish them from other operations supported by the translator. A `$[n]` in the command is replaced with the string matching the *n*th parenthetical subexpression, starting at 0 for the outer parenthesis. A `$[=]` is replaced with the text being matched, and `$[>]` is replaced with the text following the last match.

The text buffer can be replaced with an expression of the form `[`*`text`*`]`. In the example above, the `[$[1]]` following `(^(\w*) )` replaces the text buffer with the matched word.

Strings enclosed in double quotes (e.g., `"robbers"`) specify an exact match. When strings are used, they must be placed at the start of a block. NodeBrain likes to look for exact matches before regular expressions.

2.3 Rule File Using Translator Node

A translator works in concert with NodeBrain rules—hopefully not as difficult to understand as an operatic concert. Here's a rule file called `opera.nb` designed to work with the translator above.

```
#!/usr/local/bin/nb
# File: tutorial/Translator/opera.nb
define opera node translator("opera.nbx");
opera. define r1 on(info~"they didn't");
opera. define r2 if(type="HeHad");

opera:And robbers got away.
opera:And never mind.
opera:And never worry.
opera("translate"):opera.txt
```

The first highlighted line defines a translator node that uses the `opera.nbx` translator. The two rules, `r1` and `r2`, do nothing except demonstrate a rule can fire in response to foreign text.

The second highlighted line, which starts with `opera:`, sends foreign text to the translator node for translation. The last highlighted line, starting with `opera(`, directs the translator nodes to translate the foreign text file `opera.txt`.

2.4 Lost in Translation

When you execute `opera.nb`, you should see something like this.

```
$ ./opera.nb
2009/01/28 17:49:27 NB000I Argument [1] ./opera.nb
> #!/usr/local/bin/nb
> # File: tutorial/Translator/opera.nb
> define opera node translator("opera.nbx");
2009/01/28 17:49:27 NB000I Loading translator "opera.nbx"
---------- --------
# File: tutorial/Translator/opera.nbx
#

# Ignore lines starting with "#"
(^#)

# Pick up lines with "had" followed by a word,
# but continue on looking for other matches
# even when a match is found.
@(had (\w*)):alert type="Had",what="$[1]";

# Look for "he had" lines - will be subset of "had" lines
(he had (\w*)):alert type="HeHad",what="$[1]";

# Look for lines starting with "And".
(^\s*And ){
  (^then they ){
    (thought)
    :assert info="$[=]";
    }
  (^(\w*) )[$[1]]{
    "robbers":assert info="bad guys $[>]";
    "never":assert info="they didn't $[>]";
    }
  }

# Look for "He <verb>" lines.
(^He (crept|ran) ){
  @"fourteen miles in fifteen days,":assert info="slow runner this man who had naught";
  :assert heVerb="$[1]";
  }
---------- --------
2009/01/28 17:49:27 NB000I Translator "opera.nbx" loaded successfully.
```

. . . continued on next page . . .

```
> opera. define r1 on(info~"they didn't");
> opera. define r2 if(type="HeHad");
> opera:And robbers got away.
> opera. assert info="bad guys got away.";
> opera:And never mind.
> opera. assert info="they didn't mind.";
2009/01/28 17:49:27 NB000I Rule opera.r1 fired
> opera:And never worry.
> opera. assert info="they didn't worry.";
> opera("translate"):opera.txt
---------- --------> opera.txt
> opera. alert type="Had",what="naught";
> opera. alert type="HeHad",what="naught";
2009/01/28 17:49:27 NB000I Rule opera.r2 fired
> opera. assert info="bad guys came to rob him;";
> opera. assert heVerb="crept";
> opera. alert type="Had",what="him";
> opera. assert info="could not find him;";
> opera. assert info="slow runner this man who had naught";
> opera. assert heVerb="ran";
> opera. assert info="they didn't looked behind him.";
2009/01/28 17:49:27 NB000I Rule opera.r1 fired
---------- --------< opera.txt
2009/01/28 17:49:27 NB000I Source file "./opera.nb" included. size=237
2009/01/28 17:49:27 NB000I NodeBrain nb[16261] terminating - exit code=0
$
```

As an exercise, you should perform the translation yourself to make sure you understand what is going on here. At least focus on the lines around the first firing of rule `opera.r1`, starting with `opera:And never mind.` Walk `And never mind.` through the translator to see why it emits `assert info="they didn't mind."` triggering `r1`. Do you understand why `opera:And never worry.` emits a similar assertion without trigging `r1`? It is a characteristic of NodeBrain's `on` rule and the fact that the condition didn't change. An `if` rule would have fired both times.

The translator `opera.nbx` is designed to operate on lines like those found in `opera.txt`, and the rule file `opera.nb` is designed to work with the translator. You may have noticed that the translator was not designed specifically to work with these rules, because it emits commands that provide information the rules don't need. In many cases, a translator and a set of rules are designed together and the translator only emits information that the rules use.

3 Commands

This section describes the syntax and symantics of commands used with the Translator module.

3.1 Define

A translator is specified in a translator file, but defined to NodeBrain using a `define` command.

```
define node node translator("filename");
```

Parameters	Description
filename	Name of translator file. The filename by convention has a ".nbx" suffix, but this is not required.

See "Translators" in the *NodeBrain Language Reference* for information on coding translator files.

3.2 Assertions

Strings may be asserted to a translator node for translation. A translator node accepts any number of string arguments. When multiple arguments are asserted, they are translated in the order specified.

```
define term on(condition) node(string1,string2,...);     # Asserted strings
assert node(string1,string2,...);     # Asserted strings
```

3.3 Cell Expressions

Translators may be used as cell expressions for string classification.

```
define term cell node(string1,string2,...);
```

The value of a translator cell expression is determined by the translator specification file. If a "value" operator is not encountered during the string matching process, the value is 1 for recognized strings and Unknown for unrecognized strings. Other values may be explicitly specified within the translator using "value" operations of the form `$(expression)`. When value operations are encountered during the matching process, the last encountered value determines the cell expression value. If the value of the matching operation on the first argument string is Unknown, matching continues with subsequent argument strings until a value other than Unknown is established or the end of the arugment list is reached.

3.4 Node Commands

Text may be sent to a translator using a node command.

```
node:text
```

A translator node also supports special node commands identified by a "verb" argument.

```
node("verb"):body
```

A verb of "translate" instructs the node to translate a file.

```
node("translate"):filename
```

A verb of "do" is used to send a command to the translator. This can be used to update a translator dynamically in response to monitored conditions. See "Translators" in the *NodeBrain Language Reference* for information on translator commands.

```
node("do"):command
```

4 Triggers

When a string is sent to a translator node, by an assertion or a command, the node may emit NodeBrain commands. Emitted commands executed in the context of the translator node. To direct a command to another node, a context prefix is required on the command within the translator file.

Index