Tree NodeBrain Module

Release 0.9.03

Tree NodeBrain Module
December 2014
NodeBrain Open Source Projet

Release 0.9.03

Author: Ed Trettevik

History

2012-06-04 Title: *Tree NodeBrain Module*
Author: Ed Trettevik <eat@nodebrain.org>
Publisher: NodeBrain Open Source Project

2012-06-04 Release 0.8.10

- This is a first edition.

2014-12-13 Release 0.9.03

- Included prune node function and fixed some obselete statements.

Preface

This manual is intended for users of the Tree NodeBrain Module, a plug-in for storing a table of data in main memory. The reader is expected to be familiar with the basic concepts of NodeBrain. See www.nodebrain.org for general information on NodeBrain.

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

The Tree module provides a node with the skill to store information in a simple tree structure in main memory. We define a Tree node as follows.

```
define table1 node tree;
define table2 node tree;
```

When assertions are made to a Tree node, it simply stores the asserted values.

```
assert table1(1,3,5,"abc"),table1(1,3,6,"xyz");
```

To evaluate node conditions, the node module does a lookup on the tree.

```
define r1 on(x and y);
assert a=1,b=3;
assert x==table1(a,b); # x is 1 (true) because node (1,3) is present
```

You might expect the value of a Tree node condition to be False (0) when the specified node has not been asserted. Instead the value is Unknown.

```
assert b=4;    # This makes x Unknown
```

A false value can be assigned to a node.

```
assert !table1(a,b); # table1(1,4) is now False
```

To remove an entry from a tree, you assert it to be Unknown.

```
assert ?table1(a,b); # table1(1,4) is now Unknown
```

You may actually assign any value to a tree entry.

```
assert table1(a,b)=25; # table1(1,4) is now 25
assert table1(a,b)="abc"; # table1(1,4) is now .abc.
assert table1(a,b)=a+b; # table1(1,4) is now 5
assert table1(a,b)=1; # table1(1,4) is still True
assert table1(a,b)=!; # table1(1,4) is now False
assert table1(a,b)=?; # table1(1,4) is now Unknown
```

A node module is not required to fully implement the NodeBrain model. For example, the Tree module does not properly implement the association of a definition with a tree entry. That is, .assert table1(1,2,3)==a*b;. will not work the way you might hope.

We have shown how values are asserted to a table implemented by the Tree skill. Now let.s look at how we might reference the table in conditions. We just mentioned a limitation. If tree entries are given definitions, our conditions will not respond as one might hope. (This

would require a minor enhancement to our node module.) Otherwise, we can expect a tree node condition to take on the value of the currently referenced row.

```
define r1 on(table(a,b));
define r2 on(!table(b,a));
define r3 on(?table(a,b));
assert a=1,b=4;
assert table(a,b),!table(b,a); # rules r1 and r2 fire
assert ?b; # rule r3 fires
assert b=4; # rules r1 and r2 fire
assert ?table(1,4); # rule r3 fires
```

Notice that an assertion takes the values of the argument cell expressions at the time of assertion, but a condition .monitors. the value of these cells. In our example above the cell expressions are simple terms (.a. and .b.). Changes to the values of these terms change the table entry being referenced by the condition. Changes to the table entries will also change the value of the condition.

In an actual application, you might use tree node tables for classification. For example, one might create a table of web server host names and reference it in an event monitoring IF rule.

```
define webserver node tree;
assert webserver("happy.web.mycompany.com");
assert webserver("lucky.web.mycompany.com");

define r1 if(webserver(server) and event="abc"):action

# The following alert will cause r1 to fire
alert event="abc",server="lucky.web.mycompany.com";
```

You may notice that this is no different than a cache with no expiration interval. However, values can not currently be assigned to cache entries, so the following would not be possible with a cache.

```
define serverGroup node tree.tree;
assert serverGroup("lucky.web.mycompany.com")="webServer";
assert serverGroup("dns1.mycompany.com")="dnsServer";

define r1 if(serverGroup(server)="webServer" and event="abc"):action

# The following alert will cause r1 to fire
alert event="abc",server="lucky.web.mycompany.com";
```

It is also important to understand that a cache uses the .closed world. assumption, returning False when an entry is not asserted while a tree returns Unknown. A tree only returns False when an entry has been asserted to be False. For this reason, you test for existence in a tree node table using the "?" operator, but use "!" when testing for existence in a cache node table.

```
define cacheTable node cache:(s);
define treeTable node tree;

# take action if server is not in either table
define r1 if(!cacheTable(server) and ?treeTable(server)):action
```

The cache and tree skills are not aware of the operators we are using here. We are simply choosing the appropriate operator based on our awareness of what value these skills return when an entry is not defined—cache returns False (!) and tree returns Unknown (?)

2 Tutorial

> *A memory is what is left when something happens and does not completely unhappen.* —Edward de Bono (1933—)

A tree node provides NodeBrain with a place to remember things that happen until they unhappen. Actually that is true of simple cells as well. The difference is that simple cells store one value while trees store multiple associated values. So a tree can remember a more complex happening. You can think of a tree node as a table of values. Like a relational table, the values in a given row form a relation. But the table is represented using a tree-of-trees structure and a value is associated with each relation, including each left-to-right partial relation. This enable you to test for the existence of a relation or use a relation as a key for looking up a stored value.

2.1 Conditions

An expression of the form *`tree(cell1,cell2,...)`* can be used like a term in cell expressions. You can assert values for these expressions and use then as conditions. The `stuff.nb` file below illustrates these concepts.

```
#!/usr/local/bin/nb
# File: tutorial/tree/stuff.nb
define stuff node tree;
define r1 on(stuff(x,y)>17 and stuff(x,y,z)="Sally");
stuff. assert ("Kathy","apple")=2,(1,2,3)="Sally",(1,2)=27;
assert x=1,y=2;
assert z=3;
show stuff
assert ?stuff(1,2);
show stuff
assert ?stuff(1,2,3);
show stuff
assert a=stuff("Kathy","apple");
show a
```

Study the execution below. Do you know why rule `r1` fires when it does? Try rewriting the rule by replacing x, y, and z with their values at the time the rule fires. Then replace the `stuff(...)` expressions in the rule with their values at the time the rule fires.

```
$ ./stuff.nb
2014-07-20 15:26:53 NB000I Argument [1] ./stuff.nb
> #!/usr/bin/nb
> # File: tutorial/tree/stuff.nb
> define stuff node tree;
> define r1 on(stuff(x,y)>17 and stuff(x,y,z)="Sally");
> stuff. assert ("Kathy","apple")=2,(1,2,3)="Sally",(1,2)=27;
> assert x=1,y=2;
> assert z=3;
2014-07-20 15:26:53 NB000I Rule r1 fired
> show stuff
stuff = ? == node tree
  "Kathy"
    "apple"=2
  1
    2=27
      3="Sally"
> assert ?stuff(1,2);
> show stuff
stuff = ? == node tree
  "Kathy"
    "apple"=2
  1
    2
      3="Sally"
> assert ?stuff(1,2,3);
> show stuff
stuff = ? == node tree
  "Kathy"
    "apple"=2
> assert a=stuff("Kathy","apple");
> show a
a = 2
2014-07-20 15:26:53 NB000I Source file "./stuff.nb" included. size=326
2014-07-20 15:26:53 NB000I NodeBrain nb[24682] terminating - exit code=0
$
```

2.2 Watch Tables

Sometimes NodeBrain uses a tree node to specify a set of values to watch. You can use a simple tree to represent a list or a table with a single column. A multi-column table may be used if you want to watch for specific combinations of event attributes.

```
#!/usr/local/bin/nb
# File: tutorial/tree/watch.nb
define watchtype node tree;
watchtype. assert ("tree fell"),("log rolled"),("frog croaked");
define r1 if(watchtype(type));

# Sample events
alert type="tide out";
alert type="log rolled";
alert type="tide in";
alert type="frog croaked";
```

Notice how the following execution picks out the events of interest.

```
$ ./watch.nb
2014-07-20 15:30:53 NB000I Argument [1] ./watch.nb
> #!/usr/bin/nb
> # File: tutorial/tree/watch.nb
> define watchtype node tree;
> watchtype. assert ("tree fell"),("log rolled"),("frog croaked");
> define r1 if(watchtype(type));
> # Sample events
> alert type="tide out";
> alert type="log rolled";
2014-07-20 15:30:53 NB000I Rule r1 fired
> alert type="tide in";
> alert type="frog croaked";
2014-07-20 15:30:53 NB000I Rule r1 fired
2014-07-20 15:30:53 NB000I Source file "./watch.nb" included. size=283
2014-07-20 15:30:53 NB000I NodeBrain nb[24688] terminating - exit code=0
$
```

2.3 Event Sequence

I said earlier that a tree can be used to remember an event between the time it happens and unhappens. Let's look at an example that provides a better illustration of this.

```
#!/usr/local/bin/nb
# File: tutorial/tree/sequence.nb
define jumped node tree;
define r1 if(event="jump" and ?jumped(name)) jumped(name);
define r2 if(event="land" and jumped(name)) ?jumped(name);
define r3 if(event="jump" and jumped(name)): \
  $ # ${name} jumped twice without intervening landing
define r4 if(event="land" and ?jumped(name)): \
  $ # ${name} landed twice without intervening jump

# Sample events
alert event="jump",name="sally";
alert event="jump",name="joe";
alert event="land",name="sally";
alert event="land",name="joe";
alert event="jump",name="sally";
alert event="land",name="joe";
alert event="jump",name="sally";
```

In the execution below, NodeBrain was able to detect a couple things that happened and then happened again without first unhappening.

```
$ ./sequence.nb
2014-07-20 19:50:03 NB000I Argument [1] sequence.nb
> #!/usr/bin/nb
> # File: tutorial/tree/sequence.nb
> define jumped node tree;
> define r1 if(event="jump" and ?jumped(name)) jumped(name);
> define r2 if(event="land" and jumped(name)) ?jumped(name);
> define r3 if(event="jump" and jumped(name)): \
  $ # ${namei} jumped twice without an intervening landing
> define r4 if(event="land" and ?jumped(name)): \
$ # ${name} landed twice without an intervening jump
> # Sample events
> alert event="jump",name="sally";
2014-07-20 19:50:03 NB000I Rule r1 fired (jumped(name)=!!)
> alert event="jump",name="joe";
2014-07-20 19:50:03 NB000I Rule r1 fired (jumped(name)=!!)
> alert event="land",name="sally";
2014-07-20 19:50:03 NB000I Rule r2 fired (jumped(name)=?)
> alert event="land",name="joe";
2014-07-20 19:50:03 NB000I Rule r2 fired (jumped(name)=?)
> alert event="jump",name="sally";
2014-07-20 19:50:03 NB000I Rule r1 fired (jumped(name)=!!)
> alert event="land",name="joe";
2014-07-20 19:50:03 NB000I Rule r4 fired
: # joe landed twice without an intervening jump
> alert event="jump",name="sally";
2014-07-20 19:50:03 NB000I Rule r3 fired
: # sally jumped twice without an intervening landing
2014-07-20 19:50:03 NB000I Source file "sequence.nb" included. size=641
2014-07-20 19:50:03 NB000I NodeBrain nb[8693] terminating - exit code=0
$
```

A tree node can support large trees if you have enough memory on your system. From a performance perspective, tree nodes scale well because they are organized as binary trees that are kept reasonably balanced. A lookup on a tuple (`x,y,z`) is performed as a binary search for `x`, followed by a binary search for `y` in a tree owned by (`x`), followed by a binary search for `z` in a tree owned by (`x,y`).

3 Commands

This section describes commands used with a Tree node.

3.1 Define

The `define` command is used to create a Tree node.

Syntax

treeDefineCmd ::= **define** š *term* š **node** [š*treeDef*] •
treeDef ::= **tree** [: *option*]
option ::= **order | partition**

Tree nodes are most often defined with no parameters. The `order` option is used when you plan to write the tree to a file using the `store` command. The `partition` option is used to define values for blocks of key values.

```
define tree node tree;
define tree node tree:order;       # store values in order
define tree node tree:partition;   # associate values to key ranges
```

3.2 Assert and Alert

The Tree module supports assertions within `assert` or `alert` commands.

```
assert node(arg1[,arg2,...])[=value];
```

If an unkown value ("?") is assigned, the element is removed.

A `@`prune node function may also be used in a node sentence assertion to remove all subordinate elements without removing the argument element.

```
assert node@prune(arg1[,arg2,...]);
```

3.3 Cell Expressions

When a Tree node is referenced in a cell expression, it presents the value of the specified element. The example shows a reference within a cell expression of a simple `define` statement.

```
define term cell node(arg1[,arg2,...]);  # term has value of specified element
```

3.4 Node Commands

This section describes commands implemented by the Tree module for use with defined nodes.

```
node[(arg1[,arg2,...])]:verb [arguments]
```

3.4.1 Balance

The `balance` command is used to rebalance the trees within a Tree node after `flatten` or `prune` commands have made the trees unbalanced.

```
node:balance
```

3.4.2 Flatten

The `balance` command is used to completely unbalance the trees within a Tree node so they effectively become lists.

```
node:flatten
```

3.4.3 Prune

The `prune` command is used to remove elements from a Tree node.

```
node[(arg1[,arg2,...])]:prune
```

Unlike an assertion that an element is unknown, which removes the identified element and the subordinate tree, the `prune` command only removes the subordinate tree. Using a table model, it deletes the subordinate table, emptying the cells to the right of the identified cell and deleting all rows represented by the subordinate table.

The `@prune` node function performs the same operation, and can be used on both a command and an assertion context. In a future release, more of the commands in this section will be available as node functions. Some or all of these commands will then be deprecated.

3.4.4 Store

The `store` command is used to write the current measures within a Tree node to a file in the form of assertions.

```
node[(arg1[,arg2,...])]:store filename;
```

The node name is not included in the assertions. This enables the assertions to be easily applied to a different node—perhaps not even a Tree node.

3.4.5 Trace

The `trace` command is used to toggle the `trace` option for troubleshooting.

```
node:trace [on|off]
```

3.5 Module Commands

The Tree module currently implements no module level commands.

4 Triggers

The Tree module does not implement any triggers. It is intended only for looking stuff up. See the Baseline and Cache modules as examples similar to the Tree module with triggers.

Index